UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

David Pristovnik

Paket programske opreme za vgrajeni sistem LUX9

DIPLOMSKO DELO NA UNIVERZITETNEM ŠTUDIJU

Ljubljana, 2016

UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

David Pristovnik

Paket programske opreme za vgrajeni sistem LUX9

DIPLOMSKO DELO NA UNIVERZITETNEM ŠTUDIJU

MENTOR: izr. prof. dr. Uroš Lotrič

Ljubljana, 2016

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Pripravite paket programske opreme, osnovane na operacijskem sistemu Linux, za po meri razviti vgrajeni sistem na osnovi arhitekture ARM. Paket naj vsebuje vse potrebne komponente za zagon strojne opreme, prilagojen zaganjalnik, jedro z moduli in izbrano distribucijo operacijskega sistema Linux. Paket opreme naj s podporo množici različnih perifernih naprav predstavlja temelj za nadaljnje projekte z vgrajenim sistemom.

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani David Pristovnik, sem avtor diplomskega dela z naslovom:

*Paket programske opreme za vgrajeni sistem LUX9 (*angl*. Board support package for embedded system LUX9).*

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom izr. prof. dr. Uroša Lotriča,
- so elektronska oblika diplomskega dela, naslov (slov., angl.) povzetek (slov., angl.) ter ključne besede (slov., angl.) identične s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki »Dela FRI«.

V Ljubljani, dne 15. marca 2016 Podpis avtorja:

Zahvaljujem se mentorju izr. prof. dr. Urošu Lotriču za vso pomoč, nesvete, potrpežljivost in strokovno vodenje pri izdelavi diplomskega dela.

Zahvaljujem se Ines, staršem, prijateljem in sodelavcem.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
GPL	GNU General Public License	splošno dovoljenje GNU
ARM	advanced RISC machine	napredni stroj RISC
RISC	reduced instruction set computer	računalniki z majhnim številom ukazov
CISC	complex instruction set computer	računalnik z velikim številom ukazov
SIMD	single instruction, multiple data	izvajanje enega ukaza nad več podatki
MMU	memory management unit	enota za upravljanje s pomnilnikom
DSP	digital signal processor	digitalni signalni procesor
ALU	arithmetic logic unit	aritmetično logična enota
MAC	multiplier-accumulator	množilnik s seštevalnikom
PC	program counter	programski števec
SP	Stack pointer	kazalec na sklad
LR	link register	vezni register
CPSR	current program status register	trenutni statusni register
IRQ	interrupt request	prekinitvena zahteva
FIQ	fast interrupt request	hitra prekinitvena zahteva
SoM	system-on-module	sistem na modulu
USB	universal serial bus	univerzalno serijsko vodilo
UART	universal asynchronous receiver/transmitter	univerzalni asinhroni sprejemnik/oddajnik
RAM	random access memory	bralno-pisalni pomnilnik, delovni pomnilnik
ROM	read only memory	bralni pomnilnik

SPI	serial peripheral interface	serijski periferni vmesnik
ADC	analog to digital converter	analogno digitalni vmesnik
ECC	error correction code	koda za odpravljanje napak
BIOS	basic input output system	temeljni vhodno-izhodni sistem
UEFI	unified extensible firmware interface	poenoteni razširljiv strojno-programski vmesnik
GPIO	general purpose input-output pin	splošno namenska vhodno-izhodna nožica

Povzetek

Z vpeljavo operacijskega sistema na vgrajeno napravo lahko razširimo njeno funkcionalnost in s tem pohitrimo hitrost razvoja končnega produkta. V delu je opisan prenos operacijskega sistema Linux na po meri razvit vgrajeni sistem na osnovi arhitekture ARM. V prvem delu naloge se na kratko dotaknemo zgodovine arhitekture ARM ter filozofije razvoja, nato pa se osredotočimo na osnove delovanja procesorja ARM, kar nam služi kot teoretična osnova za delo. V nadaljevanju opišemo po meri razvito družino modulov in ploščic LUX9 osnovano na procesorjih ARM. Podrobneje predstavimo modul LUX-SF9G in mikroprocesor Atmel AT91SAM9G20 na njem. Sledi glavni del naloge z umestitvijo sistema Linux v vgrajeno napravo in definiranje štirih najpomembnejših komponent priprave programskega paketa Linux. V nadaljnjih poglavjih sledijo opisi prenosa zaganjalnikov, prvo stopenjskega AT91Bootstrap, drugo stopenjskega U-Boot ter delovanje in potrebne spremembe jedra Linux. Sklop teh poglavij končamo z opisom korenskega datotečnega sistema in izgradnje distribucije Ångström. V zaključku predstavimo rezultate dela in izzive na katere smo naleteli pri izdelavi projekta.

Ključne besede: paket programske opreme, vgrajen operacijski sistem Linux, arhitektura ARM, Atmel SAM9, zaganjalnik, AT91Bootstrap, U-Boot, jedro Linux, distribucija Ångström

Abstract

By introducing an operating system to an embedded device, we can expand its functionality and thereby speed-up the development of the end-product. We describe porting of a Linux operating system to a custom developed board based on ARM architecture. First part of the thesis starts with a short introduction into the history and design philosophy of the ARM architecture and operation of ARM processors. We use this information as a theoretical background for the project. Then we continue with the description of the LUX9 family of custom modules and boards based on ARM processors. Module LUX-SF9G with microprocessor Atmel AT91SAM9G is then described more into detail. The main part of the thesis describes porting of Linux operating system on the module and defines four most important elements of porting Linux to new devices. Following chapters describe porting of bootloaders, first stage AT91Bootstrap bootloader and second stage U-Boot bootloader, and customization and adaptation of the Linux kernel. We finish this section with the description of the root filesystem and building of Ångström distribution. Results and challenges we were facing during the project are presented at the end.

Keywords: base support package, embedded Linux operating system, ARM architecture, Atmel SAM9, bootloader, AT91Bootstrap, U-Boot, Linux kernel, Ångström embedded distribution

Poglavje 1 Uvod

Vgrajeni sistemi so kombinacija stojne in programske opreme, katere namen je opravljanje ene ali več točno določenih nalog. Prisotni so na vseh področjih našega življenja od industrijskih naprav, avtomobilov, medicinskih naprav, mobilnih telefonov in nenazadnje tudi igrač. Ker je vgrajen sistem načrtovan tako, da opravlja samo določeno nalogo, ga je možno optimizirati v smislu zmanjšane porabe energije, zanesljivosti, zmogljivosti in cene.

Tako kot se razvijata področji elektronike in računalništva, se z njima spreminja in razvija tudi definicija vgrajenega sistema. Iz obdobja, ko je bil vgrajen sistem definiran kot naprava, ki opravlja eno samo namensko funkcijo in je zgrajena okrog mikrokrmilnika, ki jo upravlja za ta namen spisana strojna programska oprema, se sedaj nahajamo v času, ko od vgrajenih sistemov pričakujemo vedno več funkcionalnosti. Mikrokrmilnikom so se pri načrtovanju sistemov pridružili mikroprocesorji s priključenim zunanjim pomnilnikom in s tem omogočili, da smo dobili vgrajene naprave, ki so sposobne poganjati polnopravni operacijski sistem.

Uporaba operacijskega sistem na vgrajenih sistemih nam odpre številne dodatne možnosti. Operacijski sistem predstavlja most med uporabniško programsko opremo na eni strani in gonilniki ter strojno opremo na drugi. Zaradi podpore standardnim programskim vmesnikom in protokolom omogoča hitrejši razvoj aplikacijske programske opreme, ki sedaj zagotavlja ciljno funkcionalnost v vgrajeni napravi.

Zaradi številnih ekonomskih in tehničnih prednosti opažamo veliko porast uporabe operacijskega sistema Linux v vgrajenih sistemih. Ta trend se kaže vse od najmanjših naprav za internet stvari, mobilnih telefonov, televizorjev pa vse do multimedijskih sistemov v avtomobilih. Poleg že omenjenih prednosti uporabe operacijskega sistema v vgrajenih sistemih je za sistem Linux značilna modularna zasnova, prosto dostopna izvorna koda, razširljivost in nastavljivost. Njegova prednost je tudi v tem, da v osnovi podpira raznoliko strojno opremo in ga je mogoče prilagoditi da teče na manj zmogljivi strojni opremi.

Zaznali smo potencial in potrebo po kompleksnejših in bolj prilagodljivih napravah na tržišču vgrajenih sistemov, ki se umeščajo nad klasične mikrokrmilniške sisteme in pod nizkocenovne z vgrajenim operacijskim sistemom. Cilj je bil ponuditi vgrajeni sistem, ki ga bo mogoče uporabljati za načrtovanje vgrajenih naprav, za katere bi lahko uporabiti tudi mikrokrmilnik z

razvito namensko strojno programsko opremo. Razvili smo več družin vgrajenih modulov in ploščic pod imenom LUX9. Družina produktov LUX9 temelji na mikroprocesorjih podjetja Atmel iz družine SAM9, ki uporabljajo jedra ARM9.

Ker je vgrajen sistem definira skupek strojne in programske opreme, smo se odločili, da na naše naprave prenesemo operacijski sistem Linux ter na ta način zagotovimo ustrezno programsko podporo. Bodočim uporabnikom naših modulov smo tako omogočili čim bolj preprosto integracijo v lastne produkte. To pomeni, da lahko nemudoma pričnejo z razvojem dodane funkcionalnosti na nivoju aplikacijske programske opreme, saj je podpora strojni opremi na nivoju operacijskega sistema že zagotovljena.

Cilj diplomske naloge je predstaviti postopek in osnovne koncepte prenosa operacijskega sistema Linux, na po meri razvito strojno opremo, ki temelji na procesorju ARM. Osnovni nivoji abstrakcije programske opreme na vgrajenem sistemu so inicializacijska koda, gonilniki, operacijski sistem in na zadnje še aplikacijska programska oprema. V tej nalogi se bomo posvetili prvim trem nivojem abstrakcije.

Ker sta inicializacijska programska oprema, ki zažene jedro Linux in pa seveda tudi gonilniki, ki omogočajo komunikacijo jedra s strojno opremo kritičen in sestavni del prenosa vgrajenega operacijskega sistema, bomo pričeli s spoznavanjem arhitekture ARM v poglavju 4. To znanje je potrebno za razumevanje procesa in kode zaganjalnikov in gonilnikov, opisanih v podpoglavju 6.3. V poglavju 5 sledi kratek opis platforme LUX9, potem pa se osredotočimo na prenos in razumevanje delovanja jedra Linux ter izgradnji korenskega datotečnega sistema v poglavju 6. Na koncu pa bomo povzeli naše delo in zapisali opažanja v sklepnih ugotovitvah.

Poglavje 2 Arhitektura ARM

ARM je 32 bitna procesorska arhitektura RISC z nekaj posebnostmi, ki jo razvija podjetje ARM Holdings. Poslovni model podjetja temelji bolj na razvoju in licenciranju intelektualne lastnine, kot pa na proizvodnji in prodaji samih integriranih vezij. Podjetje licencira arhitekturo partnerjem, ki potem proizvajajo procesorje ARM ali na tej tehnologiji temelječe sisteme na čipu (angl. System on Chip, SoC). Obstajata dva glavna tipa licenc, proizvodna licenca ter licenca za arhitekturo. Proizvodna licenca vsebuje vse potrebne informacije za načrtovanje in proizvodnjo integriranih vezij, ki vsebujejo jedro ARM, in se dalje deli na licenco za jedro hard-core in soft-core. Jedro hard-core je optimizirano za določeni proizvodni proces in litografijo, medtem ko se lahko jedro soft-core, opisano v programskem jeziku Verilog, uporabi v katerem koli proizvodnem procesu, vendar je manj optimizirano. Licenca za arhitekturo omogoča lastniku licence razvoj lastnih procesorjev, kompatibilnih z arhitekturo ARM, na primer Apple A8, A9, Marvell StrongARM, Qualcomm Snapdragon [9].

Razlogi za zelo veliko popularnost arhitekture ARM in procesorjev z več kot 90 % deležem v današnjih mobilnih napravah [6] so v edinstveni kombinaciji lastnosti, ki jih premorejo. Jedra ARM so zelo preprosta v primerjavi z ostalimi splošno namenskimi procesorji, to pomeni majhno število potrebnih tranzistorjev za izdelavo in več prostora na silicijevi rezini za ostale bolj namenske komponente. Tipično lahko procesor ARM vsebuje poleg jedra ARM več perifernih krmilnikov, procesor DSP, grafični krmilnik, krmilnik zunanjega pomnilnika ter interni pomnilnik. Arhitektura sistemskih ukazov ARM kot tudi način implementacije cevovoda, sta usmerjena k zmanjšanju porabe energije, kar je kritičnega pomena predvsem za naprave, ki se napajajo baterijsko. Tukaj mislimo predvsem na vgrajene mobilne sisteme, na primer mobilne telefone in tablične računalnike. Ena izmed pomembnih značilnosti arhitekture ARM je njena modularna zasnova. Edini obvezni del procesorja ARM je celoštevilski cevovod, vse ostale komponente, vključno s predpomnilniki, enoto za upravljanje s pomnilnikom, procesorji s plavajočo vejico ter drugi koprocesorji so opcijski, kar omogoča veliko fleksibilnost pri razvoju specifičnega mikroprocesorja, zasnovanega na jedru ARM [2].

2.1 Kratka zgodovina

Zgodovina podjetja ARM Holdings se je pričela leta 1983, ko je podjetje Acorn Computers začelo z iskanjem 16 bitnega mikroprocesorja, ki bi nasledil 8 bitni procesor v njihovem prvem domačem računalniku. Po pregledu tržišča so ugotovili, da noben komercialno dosegljiv procesor ne ustreza njihovim zahtevam. Vsi tedanji 16 bitni mikroprocesorji CISC so bili počasnejši od standardnih pomnilnikov. Imeli so kompleksen nabor ukazov, ki so se lahko izvajali po več urinih period in s tem povzročili velike prekinitvene latence, čemur pa so se hoteli izogniti. Razlog je tičal v tem, da so bili ti mikroprocesorji modelirani po načelih procesorjev v mini računalnikih, sestavljeni pa so bili iz več čipov in upravljani z mikrokodo.

Odločili so se, da bodo izdelali lasten mikroprocesor po vzoru projekta Berkely RISC1, ki je pokazal, da je možno izdelati zelo preprost procesor z relativno majhnimi sredstvi, ki bi bil po zmogljivosti primerljiv s konkurenčnimi procesorji CISC. Rezultat je bil leta 1985 izdelan prvi procesor Acorn RISC Machine (kasneje preimenovan v Advanced RISC Machine) z uporabo manj kot 25.000 tranzistorji. Procesor ARM2 je kmalu nasledil in nadomestil procesor ARM1, ki je postal tudi prvi komercialno dosegljiv procesor RISC na svetu zasnovan s samo 30.000 tranzistorji. Imel je 32 bitno podatkovno vodilo in 26 bitno naslovno vodilo z 16 registri in integriranim predpomnilnikom. Prvi procesor ARM1 spada v verzijo arhitekture ARMv1, medtem ko ARM2 in ARM3, izdelan leta 1990, pa v družino arhitekture ARMv2 (tabela 1).

Istega leta se je podjetje Apple odločilo uporabiti procesor ARM v svojem produktu Newton, tako je leta 1990 nastalo skupno podjetje ARM Limited. Podjetje je izdelalo verzijo arhitekture ARMv3 z 32 bitnim naslovnim vodilom, enoto za upravljanje s pomnilnikom MMU in 64 bitnimi množilnimi ukazi ter s tem naredilo prvi korak na mobilno tržišče.

Četrta generacija arhitekture ARM je ugledala luč sveta leta 1996. Največja novost glede na prejšnje verzije je bila vpeljava 16 bitnih ukazov Thumb. Koda Thumb zavzame 40 % manj pomnilniškega prostora kot 32 bitna ARM, vendar je manj učinkovita. Najbolj znan predstavnik četrte generacije arhitekture ARM je jedro ARM7TDMI, ki še vedno ostaja številčno najbolj prodajano jedro do tega dne, najdemo ga recimo v najmanjših predvajalnikih iPod. Zanimivo, da je jedro ARM7TDMI zasnovano na prvotnem 3 stopenjskemu cevovodu kot prvi ARM iz leta 1985 in vsebuje samo 30.000 tranzistorjev.

Leta 1999 je bila predstavljena peta generacija arhitekture ARM, ki je dodala podporo za digitalno procesiranje signalov in izvajanje javanske kode. Najbolj znana predstavnika te generacije sta procesorja Intel XScale in jedro ARM926EJ-S, na katerem temelji tudi v tej nalogi opisana družina izdelkov LUX9.

V šesti generaciji arhitekture ARM iz leta 2001 so dodali ukaze SIMD, izboljšali ukaze Thumb, ter dodali tehnologijo TrustZone za virtualizacijo ter podporo za procesorje z več jedri.

Sedma generacija arhitekture ARM, je izboljšala ukaze SIMD in vpeljala enoto NEON za pospeševanje multimedije ter izboljšala podporo za računanje s plavajočo vejico. Spremenjeno je tudi poimenovanje procesorjev iz klasične oblike ARM{x}{y}{z} (na primer ARM926EJ-S) v družino jeder Cortex.

Družina Cortex-A ali aplikacijska družina procesorjev je namenjena poganjanju kompleksnih operacijskih sistemov in se uporablja v pametnih telefonih, tablicah in prenosnikih. Procesorji namenjeni poganjanju realno časovnih operacijskih sistemov so iz družine Cortex-R. V mikrokrmilnikih in napravah z zelo nizko porabo energije pa najdemo procesorje Cortex-M.

Zadnja, osma generacija arhitekture ARM vpeljuje 64 bitne ukaze, tako da ohranja združljivost in podporo za 32 bitno arhitekturo, ki bo omogočila preboj arhitekturi ARM tudi na druga področja kot so strežniki in podatkovni centri.

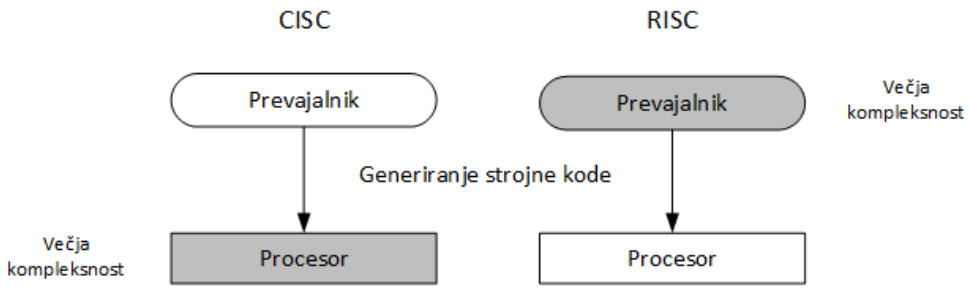
V tabeli 1 je predstavljen razvoj arhitekture ARM skozi različna obdobja s tipičnimi predstavniki jeder in lastnostmi, ki so bile dodane osnovni arhitekturi [9].

Tabela 1: Zgodovina razvoja ARM arhitekture.

Verzija arhitekture	Leto	Lastnosti	Jedra
v1	1985	Prvi procesor	ARM1
v2	1987	Podpora za koprocesor, integriran predpomnilnik	ARM2, ARM3
v3	1992	32 bit, MMU, 64 bit MAC	ARM6, ARM7
v4	1996	ukazi Thumb	ARM7TDMI, ARM9TDMI, ATM720T, ARM940T, StrongARM
v5	1999	izboljšave ARM/Thumb, ukazi DSP, razširitve Jazelle	ARM10, ARM9EJ-S, ARM926EJ-S, ARM1026EJ-S
v6	2001	ukazi SIMD, Thumb2, Trust-Zone, podpora večim jedrom	ARM11, ARM11 MPCore
v7	2005	NEON, izboljšan MMU, podpora za dinamične vejitve	Cortex-A5, Cortex-A9, Cortex-A15, Cortex-R, Cortex-M
v8	2011	Dodana 64bit arhitektura imenovana AArch64 in definirani novi sistemski ukazi A64	Cortex-A35, Cortex-A53, Cortex-A57

Poglavje 3 Pregled arhitekture ARM in filozofija razvoja

Jedro ARM arhitekturno spada med procesorje RISC. Filozofija razvoja arhitekture RISC si prizadeva zagotoviti preproste vendar uporabno močne ukaze, ki se izvedejo v eni urini periodi pri visokih frekvencah delovnega takta. Osnovno načelo te filozofije predvideva zmanjšanje kompleksnosti ukazov, ki se izvajajo na strojnem nivoju, ker je lažje doseči večjo fleksibilnost in inteligenco v programski opremi kot pa v strojni. Posledično je kompleksnost in razvoj prevajalnikov zahtevnejša. Nasprotno pa se klasična arhitektura CISC bolj opira na strojno opremo glede zagotavljanja funkcionalnosti ukazov, kar prikazuje slika 1[12].



Slika 1: Primerjava arhitektur CISC in RISC.

Smernice arhitekture RISC narekujejo razvoj procesorjev ob upoštevanju štirih glavnih načel:

- **Ukazi** – procesorji RISC imajo zmanjšano število skupin ukazov. Te skupine predstavljajo posamezne preproste ukaze, ki se vsak lahko izvede v eni urini periodi. Prevajalnik se potem na višjem nivoju ukvarja s sintezo kompleksnejših operacij z nizanjem preprostih ukazov. Vsak ukaz je fiksne dolžine, kar omogoča cevovodu, da prične z nalaganjem novega ukaza, še preden je bil trenutni dekodiran.
- **Cevovodi** – ukazi se razbijejo na manjše enote, ki se potem izvajajo paralelno v cevovodih. V idealnem primeru se za doseganje maksimalne prepustnosti vsaka stopnja cevovoda izvede v enem urinem ciklu. Ukazi se lahko dekodirajo v eni stopnji cevovoda, s tem tudi ni potrebe po mikroprogramiranju za dekodiranje ukazov kot pri arhitekturi CISC.

- **Registri** – procesorji RISC imajo veliko število splošno namenskih registrov, katerih vsebina je lahko ali podatek ali naslov. Registri služijo kot hitri lokalni pomnilnik za vse operacije na podatkih, za razliko procesorji CISC poznajo več vrst registrov.
- **Arhitektura naloži-shrani (angl. load-store)** – procesor izvaja operacije na podatkih v registrih, posebni ukazi naloži-shrani se uporabljajo za prenos podatkov med registri in zunanjim pomnilnikom. Procesorji CISC lahko izvajajo operacije na lokacijah direktno v pomnilniku.

Tako kot vsi ostali procesorji RISC, tudi procesorji ARM temeljijo na arhitekturi naloži-shrani kar pomeni, da ukazi, ki so namenjeni manipulaciji podatkov delujejo samo na registrih in so ločeni od tistih za upravljanje s pomnilnikom. Vsi ukazi ARM so 32 bitni in večina s tremi operandi. Za arhitekturo ARM je tudi značilno 16 splošno namenskih 32 bitnih registrov. Vse te lastnosti določajo in definirajo za arhitekturo ARM značilni cevovod, ki pa se od verzije do verzije arhitekture in izpeljanke jedra lahko precej razlikuje.

Kljub želji po upoštevanju vseh načel arhitekture RISC in potrebi po čim bolj preprosti implementaciji se nekaj razvojnih odločitev ne ujema s to filozofijo. Originalni Berkeley načrti za procesor RISC predvidevajo uporabo registrskih oken za pospešitev preklapljanja med procesi, vendar so bila opuščena zaradi prevelike kompleksnosti. Klasični pristop RISC tudi predvideva izvajanje vsake stopnje cevovoda v eni urini periodi, kar je pomembno pri načrtovanju 3 stopenjskega cevovoda. Večina ukazov se res izvede v eni urini periodi z izjemo ukazov za delo z zunanjim pomnilnikom. Dokončanje preprostega ukaza LOAD ali STORE v eni urini periodi bi pomenilo dva dostopa do pomnilnika: enega za prenos naslednjega ukaza iz pomnilnika in drugega za dejanski prenos. Dvojni dostop do pomnilnika v eni urini periodi zahteva Harvardsko arhitekturo z ločenim pomnilnikom za ukaze in operande, kar se je za takratne čase zdelo predrago in je privedlo do uvedbe načina naslavljanja z avtomatskim indeksiranjem, kjer se vrednost indeksnega registra poveča ali zmanjša medtem ko se izvaja operacija naloži ali shrani. Čeprav vse arhitekture ARM od verzije 5 naprej uporabljajo ločene predpomnilnike za ukaze in operande, da lahko naredijo prenos v eni urini periodi, za zagotavljanje večje zmogljivosti in optimizacije velikosti kode še vedno podpirajo ta način delovanja. Posebnost arhitekture ARM so tudi ukazi za več registrske prenose, ki omogočajo, da se podatki naložijo v 16 registrov naenkrat ali pa obratno. Čeprav ti ukazi potrebujejo več kot eno urino periodo, so zelo učinkoviti pri pospeševanju zmogljivostno občutljivih operacij, kot so večji prenosi podatkov.

Jedro ARM ni čista izvedba arhitekture RISC zaradi omejitev, ki jih narekuje njihova primarna uporaba - za vgrajene (angl. embedded) sisteme. Ugotovimo lahko, da je ravno to prednost jedra ARM. V današnjih sistemih ni ključ do uspeha v čisti in največji procesorski moči - hitrosti,

temveč je pomembna zmogljivost sistema kot celote vključno s porabo energije. Lep primer tega je podjetje Intel, ki prevladuje na tržišču visoko zmogljivih procesorjev za strežnike, namizne računalnike in prenosnike. Že nekaj časa se trudi prenesti svojo arhitekturo x86 tudi v mobilne naprave vendar mu, predvsem zaradi prevelike porabe in posledično premajhne avtonomije naprav, to ne uspeva.

3.1 Nabor ukazov za vgrajene sisteme

Nabor ukazov ARM se razlikuje od čistih RISC glede na potrebe in primernost za vgrajene sisteme:

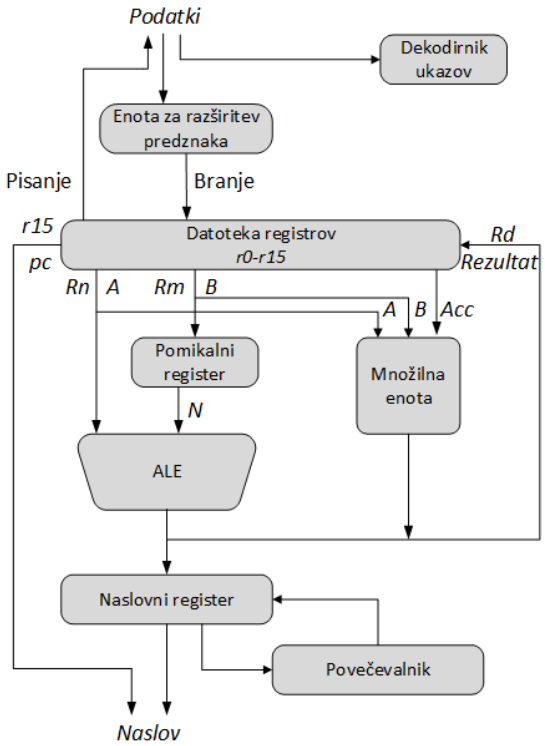
- **Variabilno trajanje izvajanja določenih ukazov** – vsak ukaz se ne izvede v eni urini periodi. Na primer čas izvajanja ukazov LOAD-STORE-MULTIPLE je odvisen od števila registrov, ki jih je potrebno prenesti v pomnilnik. Prenosi se izvajajo na zaporednih pomnilniških naslovih, kar povečuje zmogljivost, ker je zaradi narave delovanja dinamičnega pomnilnika dostop do zaporednih naslovov hitrejši od naključnega dostopa. S tem je povečana tudi gostota kode, saj je večje število prenosov registrov značilno za začetek in konec izvajanja funkcij.
- **Pomikalni register (angl. inline barrel shifter)** – pomikalni register je strojna komponenta, ki predprocesira in pripravi enega od vhodnih registrov, preden ga lahko ukaz uporabi. Ta posebnost omogoča razširitve ukazov za povečanje zmogljivosti in povečanje gostote kode.
- **16 bitni ukazi Thumb** – ARM je razširil jedro s tem, da je dodal še skupino 16 bitnih ukazov Thumb. Ti ukazi povečajo gostoto kode za približno 30 %, z manjšim pribitkom na hitrosti izvajanja, kar pa je pomembno predvsem za naprave, ki imajo omejeno velikost pomnilnika. Ukaze Thumb smo bili prisiljeni uporabiti pri programiranju prvostopenjskega zaganjalnika za mikrokrmilnik Atmel AT91SAM9260, ker ima na razpolago samo 4 KB internega pomnilnika SRAM.
- **Pogojno izvajanje** – ukaz je izveden samo takrat, ko je izpolnjen določen pogoj. Ta funkcionalnost izboljša zmogljivost in poveča gostoto kode s tem, da zmanjša število vejitev.
- **Izboljšani ukazi** – k standardnemu ukaznemu naboru ukazov ARM so dodali ukaze za digitalno procesiranje signalov (angl. Digital Signal Processing, DSP), ki podpirajo hitro izvajanje 16x16 bitnih množenj. Ti ukazi omogočajo, da lahko hitrejši procesorji ARM nadomestijo kombinacije procesorja in koprocesorja DSP.

Vse te dodatne lastnosti so omogočile, da je arhitektura ARM ena izmed vodilnih ter najbolj uporabljenih in zastopanih 32 bitnih arhitektur za vgrajene in mobilne naprave na tržišču. V zadnjem času pa se prebija, zaradi svoje energetske učinkovitosti, tudi v svet strežnikov v velikih podatkovnih centrih.

Poglavje 4 Osnove procesorja ARM

Programer si lahko jedro ARM predstavlja kot skupek funkcijskih enot, povezanih s podatkovnim vodilom. Podatki vstopijo v jedro preko podatkovnega vodila. Podatek je lahko ukaz, ki se bo dekodiral in izvedel ali pa operand nad katerim se izvajajo operacije. Na sliki 2 je predstavljena Von Neumannova izvedba arhitekture ARM, značilna za prve verzije jeder, katere tipičen predstavnik je ARM7TDMI. Puščice predstavljajo tok podatkov, povezave podatkovno vodilo, pravokotniki pa enote, ki opravljajo določeno operacijo ali pa pomnilniško funkcijo. Vse to skupaj na abstrakten način predstavlja komponente, ki sestavljajo jedro ARM[12].

Harvardska arhitektura, ki se uporablja v kasnejših verzijah, uporablja dve podatkovni vodili, eno za ukaze in drugo za operande. Za programerja je pomembno osnovno poznavanje arhitekture in principov delovanja, zato zadošča, da si koncepte ogledamo na osnovni verziji jedra.



Slika 2: Tok podatkov v jedru ARM.

Dekodirnik ukazov prevede ukaze preden se izvedejo, vsak ukaz, ki se izvede spada v neko množico ukazov. Procesor ARM, kot vsi ostali procesorji RISC, uporablja arhitekturo naloži-shrani. To pomeni, da ima dva tipa ukazov za prenos podatkov v in iz procesorja. Ukazi LOAD prekopirajo podatke iz pomnilnika v registre jedra, medtem ko ukazi STORE skopirajo podatke iz registrov nazaj v pomnilnik. Za arhitekturo RISC je značilno, da ne obstajajo ukazi, ki bi direktno manipulirali z lokacijami in podatki v pomnilniku. Operacije se zato izvajajo samo na podatkih v registrih.

Enote podatkov se iz pomnilnika prenesejo v datoteko registrov, ki jo sestavlja množica 32 bitnih registrov. Ker je jedro ARM 32 bitni procesor, večina ukazov predvideva, da registri vsebujejo predznačene ali nepredznačene 32 bitne vrednosti. Enota za razširitev predznaka avtomatsko pretvori 8 in 16 bitna predznačena števila v 32 bitne vrednosti, ko se preberejo iz pomnilnika v registre.

Ukazi ARM tipično vsebujejo dva izvorna registra Rn in Rm ter en ciljni register Rd. Izvorni operandi se prenesejo iz internih registrov po vodilih A in B.

Aritmetično logična enota (ALE) ali množilna enota (MAC) vzameta vrednosti Rn in Rm iz vodila A in B in izračunata rezultat. Ukazi za obdelavo podatkov rezultat Rd zapišejo nazaj v registre. Ukazi LOAD in STORE uporabijo ALE enoto za generiranje naslovov, ki se nahajajo v naslovnem registru, ter jih postavijo na naslovno vodilo.

Posebnost jedra ARM je, da se lahko podatki iz registra Rm še dodatno obdelajo v pomikalnem registru preden preidejo v enoto ALE. Skupaj tako lahko pomikalni register in enota ALE izračunata širok spekter izrazov in naslovov.

Po prehodu skozi funkcijske enote, se rezultat operacij Rd zapiše nazaj v registre preko vodila za rezultate. Za ukaze LOAD in STORE povečevalnik (angl. incrementer) osveži naslovni register, preden jedro prebere ali zapiše novo vrednost registra iz ali na naslednji zaporedni naslov v pomnilniku. Procesor izvaja ukaze dokler izjema ali prekinitev ne spremeni normalnega toka izvajanja.

4.1 Registri

Splošno namenski registri vsebujejo ali podatke ali pa naslove in so 32 bitni. Označeni so s črko r, ki ji sledi številka registra, na primer, register 4 je označen z r4. Slika 3 prikazuje aktivne registre, ki so na voljo v uporabniškem načinu – to je zaščiten način v katerem tečejo programi. Procesor lahko deluje v sedmih različnih načinih.

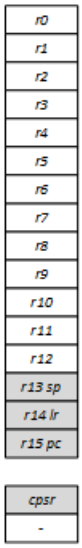
Na voljo je do 18 aktivnih registrov: 16 podatkovnih in 2 statusna registra. Podatkovni registri so programerju vidni kot r0 do r15. Procesor ARM ima tri registre, ki so namenjeni posebni funkciji, to so r13, r14 in r15 in se jih navadno drugače označuje glede na njihovo namembnost:

- r13 se uporablja kot kazalec na sklad (angl. stack pointer, SP) in kaže na začetek sklada v trenutnem načinu;
- r14 se imenuje vezni register (angl. link register, LR), kjer se nahaja povratni naslov za vrnitev iz podprograma;
- r15 se uporablja kot programski števec (angl. program counter, PC) in vsebuje naslov naslednjega ukaza za prenos.

Odvisno od konteksta se lahko registra r13 in r14 uporabita tudi kot splošno namenska registra, kar je lahko še posebej uporabno, ker se vsebina teh registrov shrani med zamenjavo načina (konteksta) delovanja procesorja. Če na procesorju teče operacijski sistem, je uporaba registra r13 lahko nevarna, saj jih večina pričakuje, da r13 vedno kaže na začetek sklada.

V načinu delovanja ARM so registri r0 do r13 med seboj ortogonalni, saj se katerikoli ukaz, ki se lahko izvede nad r0, lahko enakovredno izvede tudi nad ostalimi registri. Obstajajo pa ukazi, ki obravnavajo registra r14 in r15 na poseben način.

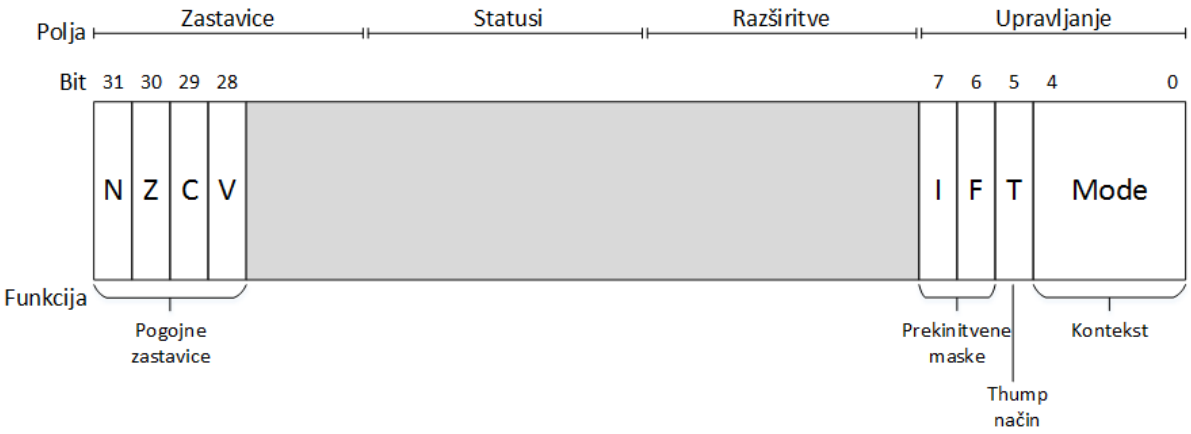
Poleg teh 16 podatkovnih registrov, obstajata tudi dva statusna registra: cpsr in spsr (trenutni in shranjeni statusni register). Datoteka registrov vsebuje vse registre vidne programerju in so odvisni od trenutnega načina delovanja v katerem je procesor.



Slika 3: Registri v uporabniškem načinu.

4.2 Trenutni statusni register (CPSR)

Trenutni statusni register (angl. current program status register, cpsr) je namenjen nadzoru in upravljanju internih operacij jedra. Gre za 32 bitni namenski register prikazan na sliki 4. Register cpsr je razdeljen na štiri dele velikosti 8 bitov, ki jih delimo na zastavice (angl. flags), statuse (angl. status), razširitve (angl. extension) in upravljanje (angl. control). Upravljavsko polje vsebuje procesorski način delovanja (kontekst), maske bitov stanja in prekinitev. Polje zastavic vsebuje bite, ki predstavljajo pogojne zastavice. Nekatera procesorska jedra ARM imajo uporabljene dodatne bite, na primer bit J v polju zastavic, ki je prisoten samo na procesorjih s podporo za tehnologijo Jazelle [7].



Slika 4: Generični statusni register.

4.2.1 Procesorski načini delovanja (kontekst)

Procesorski načini delovanja določajo kateri registri so aktivni ter pravice nivoja dostopa do cpsr registra. Privilegiran način omogoča polni bralno pisalni dostop do cpsr registra, medtem ko neprivilegiran način omogoča branje kontrolnih polj, pisanje pa je omogočeno na polju pogojnih zastavic. Skupaj obstaja sedem načinov delovanja procesorja, od tega šest privilegiranih ter en neprivilegiran način (uporabniški ali user način delovanja).

Procesor preide v prekinitveni način delovanja abort, če ne more dostopati do pomnilnika. Oba preostala prekinitvena načina (fast interrupt request in interrupt request) pa opisujeta dva nivoja prekinitev, ki jih pozna procesor ARM. Nadzorni (supervisor) način se uporablja ob zagonu procesorja in običajno v tem nivoju teče tudi jedro operacijskega sistema. Sistemski način (system) je posebna verzija uporabniškega načina, ki omogoča poln dostop do registra cpsr. Nedefiniran (undefined) način pa se uporablja takrat, ko procesor naleti na ukaz, ki ni definiran

ali pa ni podprt v tisti verziji jedra. Uporabniški način delovanja je namenjen poganjanju programov.

4.2.2 Skriti registri

Jedro ARM vsebuje vsega skupaj 37 registrov, od tega jih je 20 skritih pred programom, odvisno od stanja in načina delovanja procesorja v različnem času. Ti registri se imenujejo skriti registri (angl. banked registers) in so na sliki 5 označeni s sivo barvo. Dosegljivi so samo takrat, ko je procesor v določenem načinu delovanja. Na primer prekinitveni način delovanja abort ima registre r13_abt, r14_abt in spsr_abt skrite. Skriti registri so označeni s podčrtajem in imenom načina delovanja.

Privilegiran način delovanja (Privileged)

Prekinitveni način delovanja (Exception)

User	System	Fast interrupt	Interrupt	Supervisor	Undefined	Abort
r0	r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7	r7
r8	r8	r8_fiq	r8	r8	r8	r8
r9	r9	r9_fiq	r9	r9	r9	r9
r10	r10	r10_fiq	r10	r10	r10	r10
r11	r11	r11_fiq	r11	r11	r11	r11
r12	r12	r12_fiq	r12	r12	r12	r12
r13 sp	r13 sp	r13_fiq	r13_irq	r13_svc	r13_und	r13_abt
r14 lr	r14 lr	r14_fiq	r14_irq	r14_svc	r14_und	r14_abt
r15 pc	r15 pc	r15 pc	r15 pc	r15 pc	r15 pc	r15 pc
cpsr	cpsr	cpsr	cpsr	cpsr	cpsr	cpsr
		spsr_fiq	spsr_irq	spsr_svc	spsr_und	spsr_abt

Siva barva pomeni, da je bil register iz uporabniškega ali sistemskega načina delovanja zamenjan z registrom iz prekinitvenega načina delovanja.

Slika 5: Povezava registrov in načinov delovanja procesorja ARM.

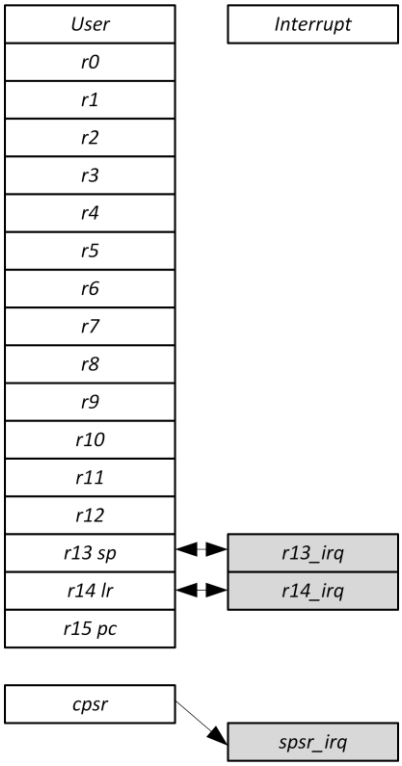
Način delovanja procesorja lahko spremenimo s spremembo ustreznih bitov v statusnem registru cpsr. Vsi procesorski načini z izjemo sistemskega imajo skupek skritih registrov, ki so

podmnožica glavnih 16 registrov. Skriti registri so preslikava uporabniških registrov. Če se spremeni način delovanja procesorja, bo skriti register nadomestil starega iz prejšnjega načina.

Ko je procesor v prekinitvenem načinu delovanja, ukazi še vedno dostopajo do registrov z imenom r13 in r14, vendar so to v realnosti skriti registri r13_irq in r14_irq. Vrednosti registrov uporabniškega načina r13_usr in r14_usr ostanejo nedotaknjeni, program pa ima še vedno normalen dostop do registrov r0 do r12.

Način delovanja procesorja lahko program spremeni tako, da zapiše nov način direktno v cpsr register (procesorsko jedro mora biti v privilegiranem načinu delovanja) ali pa strojna oprema, ko se jedro odzove na prekinitev ali past. Te prekinitve in pasti so v tehnični dokumentaciji poimenovane kot: reset, interrupt request, fast interrupt request, software interrupt, data abort, prefetch abort in undefined instruction. Pasti in prekinitve prekinejo normalno izvajanje procesorja in skočijo na predvideno pomnilniško lokacijo.

Na sliki 6 je predstavljeno kaj se zgodi, ko prekinitev povzroči spremembo načina delovanja procesorja. Predstavljena je sprememba iz uporabniškega načina delovanja v prekinitvenega, ko se ob sprožitvi prekinitve zunanje naprave zgodi prekinitvena zahteva v jedru procesorja. Takrat se uporabniška registra r13 in r14 skrijeta, zamenjajo pa ju registri r13_irq in r14_irq. Register r14_irq vsebuje povratni naslov, r13_irq pa kazalec na sklad v prekinitvenem načinu delovanja. V tem načinu delovanja lahko opazimo tudi nov register spsr ali shranjeni programski statusni register, v katerega se shrani prejšnje stanje registra cpsr. Na sliki je vidno, kako se vsebina registra cpsr prekopira v register spsr_irq. Povratek nazaj v uporabniški način omogoča poseben ukaz, ki jedru naroči, da vzpostavi prvotno stanje registra cpsr s pomočjo kopije stanja registra spsr_irq in skrije registra r13 in r14 prekinitvenega načina. Register spsr se lahko bere in spreminja samo v privilegiranem načinu, medtem ko v uporabniškem načinu ne obstaja.



Slika 6: Sprememba načina delovanja ob prekinitvi.

Pomembno je poudariti, da se vsebina registra cpsr ne kopira v register spsr, če je spremembo načina delovanja procesorja povzročil program s pisanjem v register cpsr. Vsebina registrov cpsr se shrani samo ob sprožitvi prekinitve ali pasti.

Trenutni način delovanja procesorja predstavlja pet najmanj pomembnih bitov statusnega registra cpsr. Ko se jedro zbudi oziroma dobi napajanje, se zažene v nadzornem načinu, ki je privilegiran, kar omogoča inicializacijski kodi, da nastavi kazalce na sklad v vsakem od drugih načinov. Tabela 2 predstavlja vse možne načine delovanja procesorja, ter bitni vzorec njihove predstavitve v registru cpsr.

Tabela 2: Načini delovanja procesorja ARM.

Način delovanja	Okrajšava	Privilegiran	Bitna predstavitev [4:0]
abort	abt	da	10111
fast interrupt request	fiq	da	10001
interrupt request	irq	da	10010
supervisor	svc	da	10011
system	sys	da	11111
undefined	und	da	11011
user	usr	ne	10000

4.2.3 Stanje procesorja in množice ukazov

Stanje jedra odloča kateri tip ukaznih nizov se bo izvajal. Obstajajo tri skupine ukaznih nizov: ARM, Thumb in Jazelle. Ukazi ARM so aktivni in se izvajajo samo takrat, ko je procesor v stanju ARM, enako velja za ostali dve skupini in stanji. Ko je procesor enkrat v določenem stanju, lahko izvaja samo ukaze iz tistega niza. Mešanje ukaznih nizov iz različnih skupin ni mogoče.

Biti Jazelle J in Thumb T v registru cpsr predstavljajo v katerem stanju je procesor. Če sta oba bita postavljena na 0, potem je procesor v stanju ARM in izvaja ukaze iz skupine ukaznih nizov ARM. To je tudi stanje v katerem se procesor prebudi. Če je bit T postavljen na 1, potem procesor izvaja 16 bitne Thumb ukaze. Pri spremembi stanja jedro izvede poseben vejitveni ukaz.

4.2.4 Maskiranje prekinitev

S prekinitvenimi maskami je možno ustaviti proženje določenih prekinitev. Obstajata dva nivoja prekinitev: prekinitvena zahteva (IRQ) in hitra prekinitvena zahteva (FIQ). Maskiranje je možno s postavitvijo bitov 6 in 7 v registru cpsr.

4.2.5 Pogojne zastavice

Biti v registru cpsr, ki predstavljajo pogojne zastavice se posodobijo ob izvedbi primerjave ali pa če je bil nastavljen bit S za ALE operacijo. Če je na primer rezultat ukaza za odštevanje 0, potem se postavi bit Z v registru cpsr.

Procesorska jedra, ki vsebujejo razširitve DSP, uporabljajo bit Q za označevanje, če je prišlo do prelivanja ali nasičenja pri izboljšanih ukazih DSP. Ta zastavica je lepljiva, kar pomeni, da jo strojna oprema sama postavi, za izbris pa je potrebno direktno pisanje v register cpsr.

Večina ukazov ARM se lahko izvede pogojno, glede na vrednosti pogojnih zastavic registra cpsr predstavljenih v tabeli 3 s kratkim opisom.

Tabela 3: Pogojne zastavice.

Zastavica	Ime zastavice	Pogoj za postavitev
Q	Saturation	Pri izvajanju je prišlo do prelivanja ali nasičenja
V	oVerflow	Pri izvajanju je prišlo do predznačnega prelivanja
C	Carry	Nepredznačen prenos
Z	Zero	Rezultat je 0
N	Negative	31 bit rezultata je 1

4.2.6 Pogojno izvajanje

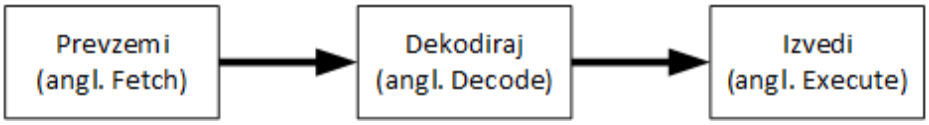
Pogojno izvajanje kontrolira ali bo jedro izvedlo določen ukaz ali ne. Večina ukazov ima pogojne atribute, ki določajo ali bo jedro ukaz izvedlo glede na postavitev pogojnih zastavic. Pred izvajanjem ukaza jedro primerja pogojni atribut s statusom pogojnih zastavic v registru cpsr in če ustrezajo potem ukaz izvede, drugače ga ignorira.

4.3 Cevovod

Kot že omenjeno je ena od pomembnih lastnosti arhitekture ARM njena preprostost. Brez uporabe registrskih oken, preimenovanja registrov, izvajanje ukazov iz vrste (do verzije jedra Cortex-A9) in drugih kompleksnih optimizacij, ki jih najdemo v sodobnih mikroprocesorjih, ostaja izvedba cevovoda ARM dokaj preprosta.

4.3.1 3 stopenjski cevovod

Uporaba cevovoda pohitri izvajanje ukazov tako, da se nov ukaz začne nalagati iz pomnilnika, medtem ko so drugi še v fazi dekodiranja in izvajanja. Za prvotne procesorje ARM je značilen preprost 3 stopenjski cevovod, ki je prikazan na sliki 7 [12]:

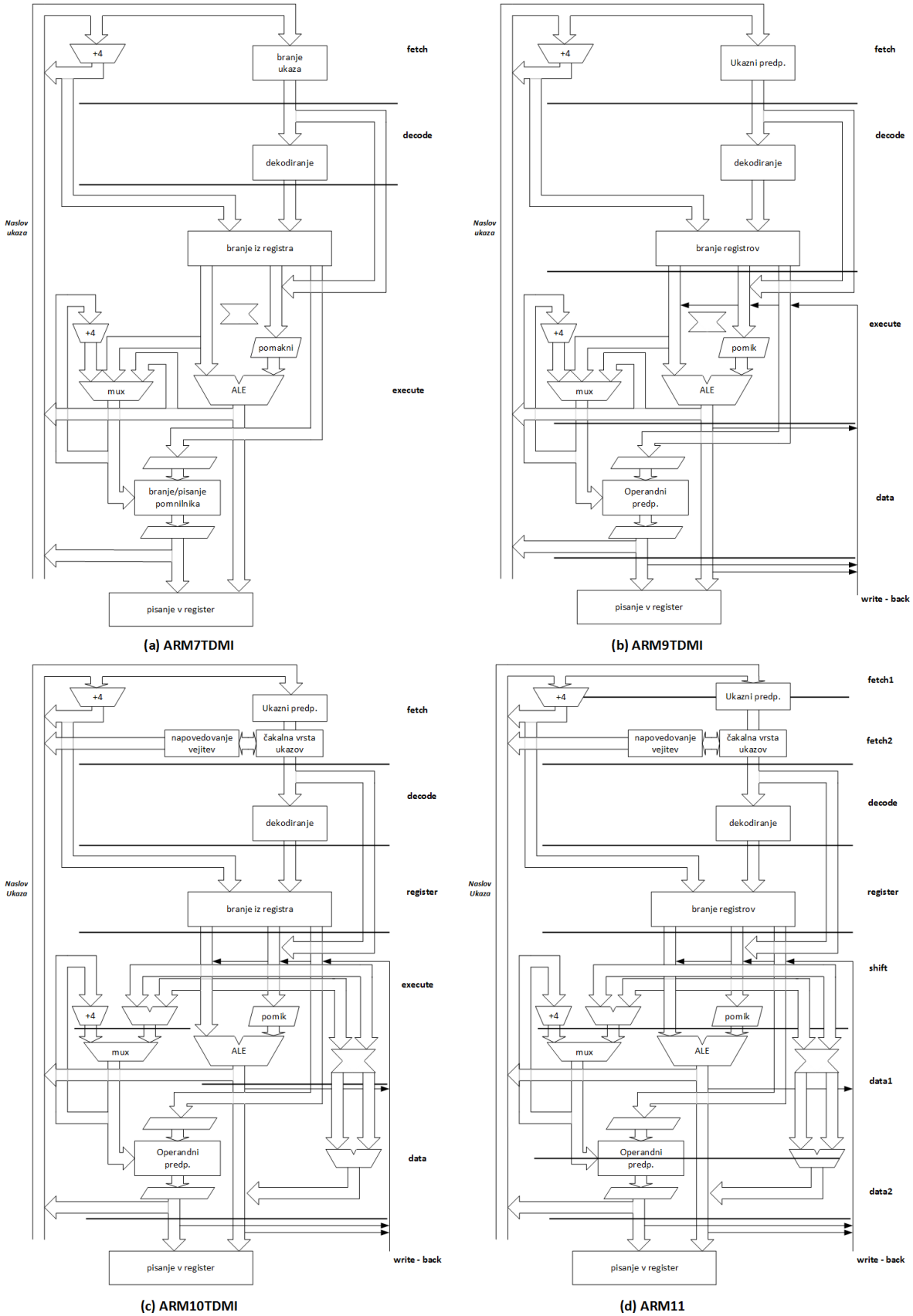


Slika 7: Preprost 3 stopenjski cevovod.

- stopnja **prevzemi** (angl. fetch) - prevzame ukaza iz pomnilnika;
- stopnja **dekodiraj** (angl. decode) - dekodira ukaz, ki ga je potrebno izvesti;
- stopnja **izvedi** (angl. execute) - izvede ukaz in zapiše rezultate nazaj v register.

Cevovod omogoča jedru, da izvede en ukaz vsako urino periodo, če ne upoštevamo cevovodnih pasti ali dostopov do pomnilnika. Prva stopnja prebere ukaz iz pomnilnika in poveča vrednost ukazno naslovnega registra, v katerem je shranjen naslov naslednjega ukaza za branje. Ta vrednost je tudi zapisana v programskem števcu. Naslednja stopnja dekodira ukaz in pripravi kontrolne signale, ki so potrebni za izvedbo ukaza. Tretja stopnja dejansko opravi vso delo, prebere operande iz registrov, izvede ALE operacije, prebere in zapiše vrednosti v pomnilnik če je potrebno in nazadnje zapiše nazaj spremenjene vrednosti v registre. Če se ukaz izvaja nad podatki, se rezultat, ki ga izračuna ALE direktno zapiše v register in se s tem izvedbena stopnja (execute) zaključi v eni urini periodi. Če gre za ukaz LOAD ali STORE, se izračunana vrednost naslova, pojavi na naslovnem vodilu, dejanski dostop do pomnilnika pa se izvede v drugi urini periodi.

Ker se programski števec (PC) lahko uporablja kot splošno namenski register, je v cevovodu predvidenih več mest kjer se lahko pojavi naslov naslednjega ukaza. V normalnem načinu delovanja, se vsebina PC poveča vsako urino periodo med stopnjo prevzemi (fetch). Če ukaz, ki se izvaja določi register r15 kot ciljni operand, potem se rezultat enote ALE uporabi za naslednji naslov ukaza. Podoben učinek ima pisanje v register r15.



Slika 8: Evolucija cevovoda ARM.

4.3.2 5 stopenjski cevovod

Če si ogledamo različne izvedbe cevovodov na sliki 8 glede na generacijo arhitekture ARM, lahko opazimo da ima ARM7TDMI in starejše arhitekture samo en predpomnilnik, kar je značilno za preproste Von Neumanove arhitekture. To pomeni, da je vsak ukaz, ki se je ukvarjal s prenosom podatkov povzročil zastoj cevovoda, ker se naslednji ukaz ne more naložiti medtem, ko se izvaja dostop do pomnilnika. Ta problem so rešili v jedru ARM9TDMI in vseh naslednjih, ki so predstavniki Harvardske arhitekture. Obstajata ločena ukazni in operandni predpomnilnik. To omogoča spremembo cevovoda tako, da se lahko izognemo zastojem cevovoda pri prenosu podatkov.

Da bi uravnovesili cevovod, so v jedru ARM9TDMI prestavili korak branja registrov v stopnjo dekodiranja (angl. decode), ker je bila ta stopnja precej krajša od izvedbene (angl. execute). Potem so razbili izvedbeno stopnjo na tri dele, prva opravlja aritmetične operacije, druga dostop do pomnilnika (ta stopnja je v mirovanju, če se izvajajo ukazi za obdelavo podatkov), tretja pa je zadolžena za pisanje rezultatov nazaj v registre. Rezultat sprememb je bolj uravnotežen 5 stopenjski cevovod, ki lahko deluje pri višjih frekvencah ure, ampak ima novo težavo – kako zagotoviti podatke stopnjam cevovoda, če so odvisne ena od druge, brez da bi bilo potrebno zaustaviti delovanje cevovoda.

4.3.3 6 stopenjski cevovod

V naslednji reviziji jedra ARM10, so ugotovili, da je zmogljivost cevovoda omejena s širino dostopa do pomnilnika, zato so razširili vodila ukaznega in podatkovnega predpomnilnika na 64 bitov. Prva stopnja cevovoda je tako lahko dostavila dva ukaza naenkrat v eni urini periodi, kar je omogočilo uvedbo statične enote za napovedovanje vejitve (angl. branch prediction). V izvedbeni stopnji je 64 bitno vodilo omogočilo izboljšanje zmogljivosti s prenosom dveh registrov naenkrat.

Izvedbena stopnja je bila razbremenjena tako, da je bil predstavljen dodaten seštevalnik za pomnoži in akumuliraj ukaze, namesto da bi se za to uporabljala glavna ALE enota. Ker ukazi za množenje ne potrebujejo dostopa do pomnilnika, je bil ta seštevalnik dodan v podatkovno stopnjo, kar je privedlo do bolj uravnoteženega cevovoda in višjih delovnih frekvenc. Dodaten seštevalnik je bil dodan še v stopnjo, kjer se dostopa do pomnilnika, ker je bila ta sedaj najbolj dolga stopnja cevovoda. Ker je izračun naslova vedno samo preprosta operacija seštevanja in je lahko ta seštevalnik opravil svojo nalogo v času manj kot v eni urini periodi, je ostala še ena in pol urine periode za dostop do pomnilnika.

Zadnja izboljšava ARM10 cevovoda je pomenila delitev dekodiranja ukazov na dve stopnji in s tem 6 stopenjski cevovod z višjo možno frekvenco delovanja.

4.3.4 8 in več stopenjski cevovod

Jedro ARM11 je bilo deležno dveh velikih sprememb pri arhitekturi cevovoda. Premikalna operacija je dobila svojo stopnjo. Podatkovni ter ukazni dostop do predpomnilnika pa sta bila razdeljena med dve ločeni stopnji.

V 8 stopenjskem cevovodu je izvajanje ukazov razdeljeno na tri različne cevovode, ki lahko delujejo vzporedno pod nekaterimi pogoji pa posredujejo ukaze v dinamičnem vrstnem redu (angl. out-of-order). Stopnje prevzemi in dekodiraj se še vedno izvajajo zaporedno (angl. in-order).

Novodobni nasledniki jedra ARM11 so Cortex-A8, A9 in A15 za katere je značilno, da imajo še višjo frekvenco delovanja in s tem tudi drugačno zasnovo cevovoda. Za Cortex-A8 je značilen 13 stopenjski statični zaporedni (in-order) cevovod za celoštevilske operacije in pa 10 stopenjski cevovod z razširitvami ukazov NEON za plavajočo vejico. Cortex-A9 ima dinamični cevovod prilagojen za dinamično z napovedovanjem (prediktivno) izvajanje ukazov. Zadnja arhitektura jedra A15 ima cevovod ločen na statični (in-order) del, ki je 12 stopenjski ter dinamični (out-of-order) del, kije dolg od 3 do 12 stopenj, odvisno od skupine ukaza, ki se trenutno izvaja [3].

4.4 Pasti in prekinitve

Ko se sproži past (angl. exception) ali pride do prekinitve (angl. interrupt), procesor nastavi vsebino PC na posebni naslov v pomnilniku. Naslov kaže na območje, ki se imenuje tabela vektorjev (angl. vector table). Tabela vektorjev vsebuje ukaze, ki kažejo na poseben podprogram, ki je namenjen temu, da obdela specifično prekinitev ali past.

V pomnilniku je za to tabelo ponavadi rezerviran naslov 0x00000000, nekateri procesorji pa imajo lahko za to tabelo rezerviran naslov 0xffff0000. Operacijski sistemi, kot je Linux, znajo dobro uporabiti to funkcijo.

Sprožitev prekinitve ali pasti ustavi normalno izvajanje ukazov procesorja in prične z nalaganjem ukazov iz tabele vektorjev pasti opisanih v tabeli 4. Vsak vnos v tej tabeli vsebuje neke vrste vejitveni ukaz, ki kaže na začetek posebnega podprograma. Arhitektura ARM določa naslednje vrste pasti oziroma prekinitev:

- **RESET** – je lokacija prvega ukaza, ki ga izvede procesor ko dobi napajanje;
- **UNDEF** – se uporabi, ko procesor ne more dekodirati ukaza;
- **SWI** – se uporabi, ko se izvede ukaz SWI, na primer pri operacijskih sistemih Linux, kot mehanizem za izvajanje sistemskih rutin;
- **PABT** – se vedno izvede v stopnji decode cevovoda, ko poskuša procesor naložiti ukaz iz naslova, brez pravih pravic dostopa;
- **DABT** – je podoben vektorju prefetch abort vector, samo da gre v tem primeru za operande;
- **IRQ** – se uporablja, če želi zunanja strojna oprema prekiniti normalno izvajanje ukazov procesorja, ob pogoju da ustrezen bit irq za prekinitve ni bil maskiran v registru cpsr;
- **FIQ** – je podoben prekinitvenemu vektorju vendar rezerviran za strojno opremo, ki potrebuje hitre odzivne čase, ponovno ob poguju da bit fiq za hitre prekinitve ni bil maskiran.

Tabela 4: Tabela vektorjev.

Kratica	Naziv pasti/prekinitve	Naslov	Višji naslov
RESET	reset	0x00000000	0xffff0000
UNDEF	undefined instruction	0x00000004	0xffff0004
SWI	software interrupt	0x00000008	0xffff0008
PABT	prefetch abort	0x0000000c	0xffff000c
DABT	data abort	0x00000010	0xffff0010
-	reserved	0x00000014	0xffff0014
IRQ	interrupt request	0x00000018	0xffff0018
FIQ	fast interrupt request	0x0000001c	0xffff001f

4.5 Komponente za razširitev jedra ARM

Standardne komponente, ki nastopajo kot razširitve jedra ARM, dodajajo nove funkcionalnosti in izboljšujejo celotno zmogljivost procesorja. Vsaka družina jedra ARM ima na voljo drugačne razširitve.

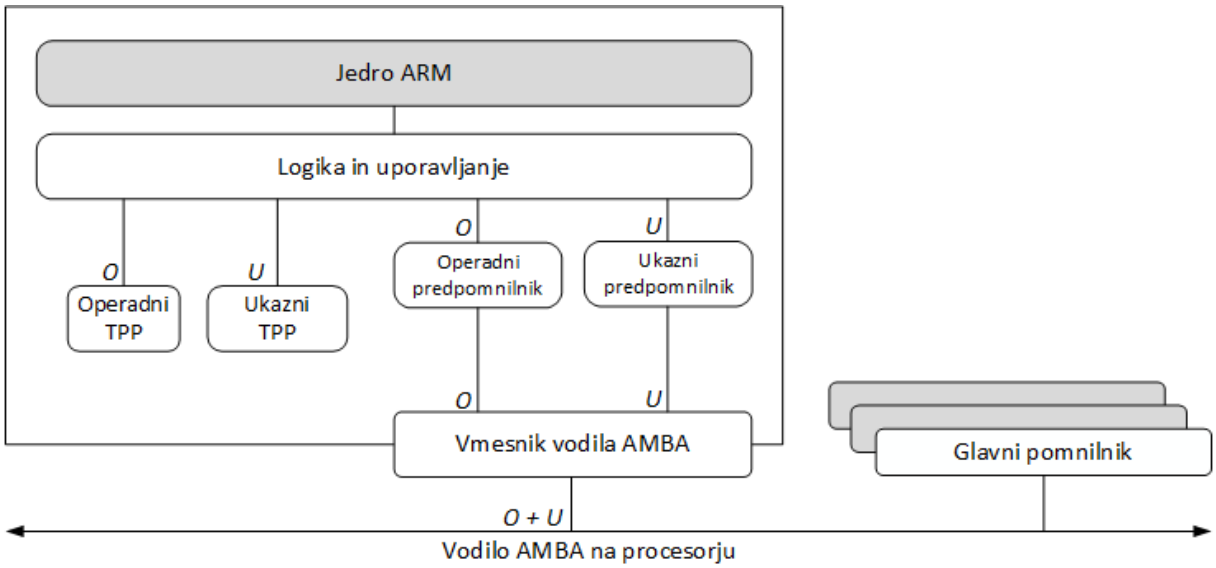
Obstajajo tri strojne razširitve, ki so tesno integrirane okrog jedra ARM: predpomnilnik in tesno povezan pomnilnik, upravljanje s pomnilnikom in pa koprocesorski vmesnik.

4.5.1 Predpomnilnik in tesno povezan pomnilnik

Predpomnilnik je blok hitrega pomnilnika postavljen med glavnim pomnilnikom in jedrom z namenom, da omogoči hitrejši dostop do podatkov. Z uporabo predpomnilnika je jedru omogočeno, da večino časa izvaja operacije brez potrebe po čakanju na podatke iz počasnega glavnega pomnilnika. Večina starejših na jedru ARM temelječih sistemov na čipu uporablja eno nivojski interni predpomnilnik. Novejši na jedru ARM11 ali Cortex-A temelječi sistemi, predvsem več jedrni, pa imajo že privzeto vgrajene dvo ali več nivojske predpomnilnike.

ARM pozna dva tipa predpomnilnika, prvi se uporablja pri Von Neumanovemu tipu jeder, kjer so v en blok pomnilnika združeni ukazi in operandi, drugi značilen za Harvardski tip jeder pa uporablja ločena predpomnilnika za ukaze in operande. Posledica uporabe predpomnilnika je povečanje zmogljivosti sistema, vendar pa se izgubi določena predvidljivost izvajanja ukazov. Vsa novejša jedra ARM od ARM9 naprej uporabljajo Harvardsko arhitekturo predpomnilnika.

Za realno časovne sisteme je determinističnost sistema, torej predvidljivost časa v katerem se bodo shranili ali naložili ukazi in operandi, zelo pomembna. To so dosegli z uporabo tesno povezanega pomnilnika (angl. tightly coupled memory). Tesno povezan pomnilnik pride v obliki hitrega SRAM pomnilnika lociranega blizu jedra, ki zagotavlja konstantno hitrost dostopa do pomnilniških lokacij za realno časovne algoritme, ki predvidevajo determinističnost sistema. Ta pomnilnik je dosegljiv na preslikanih naslovnih lokacijah. Oba tipa pomnilnika sta prikazana na sliki 9 [7].



Slika 9: Pomnilniške razširitve.

4.5.2 Upravljanje s pomnilnikom

Jedra ARM imajo tri različne tipe strojne opreme namenjene upravljanju s pomnilnikom:

- **Nezaščiten pomnilnik** je fiksen in ne omogoča fleksibilnosti. Uporablja se za majhne in preproste vgrajene sisteme, ki ne potrebujejo zaščite pomnilnika med aplikacijami.
- **Enota za zaščito pomnilnika** (angl. memory protection unit, MPU) je preprost sistem, ki uporablja omejeno število pomnilniških lokacij. Te lokacije so upravljane s pomočjo posebnih koprocesorskih registrov, ki vsaki regiji pomnilnika določa pravice dostopa.
- **Enota za upravljanje s pomnilnikom** (angl. memory management unit, MMU) je najbolj kompleksen sistem za upravljanje s pomnilnikom, ki z uporabo translacijskih tabel omogoča natančno kontrolo nad pomnilniškim prostorom. Te tabele so shranjene v glavnem pomnilniku in omogočajo preslikavo med navideznim in fizičnim naslovnim prostorom, kot tudi pravice dostopa. Enote MMU so pogoj za pravilno delovanje večopravilnih operacijskih sistemov kot je Linux.

4.5.3 Koprocesorji

Jedru ARM se lahko doda eden ali več koprocesorjev preko posebnega koprocesorskega vmesnika. Koprocesorji razširjajo procesne zmožnosti jedra, s tem da razširijo množico ukazov jedra ali pa dodajo konfiguracijske registre.

Koprocesor je dosegljiv preko skupine za razširitve namenjenih ukazov ARM. Koprocesor definira nov tip vmesnika naloži-shrani (angl. load-store), ali pa razširi množico ukazov, kot so na primer namenski ukazi za delo s plavajočo vejico. Novi ukazi se obdelajo v fazi dekodiranja ukazov cevovoda in če ta stopnja ugotovi, da gre za koprocesorski ukaz ga posreduje ustrezni enoti. Če koprocesor ni prisoten ali pa ne prepozna ukaza, potem se sproži past za neznan ukaz, kar omogoča programsko emulacijo[7].

4.6 Razširitve ukazov ARM

4.6.1 Thumb

Ukazi Thumb so bili prvič predstavljeni v četrti reviziji arhitekture z namenom doseči večjo gostoto kode za vgrajene naprave. Ukazi predstavljajo podmnožico najbolj uporabljenih 32 bitnih ukazov ARM, stisnjenih v 16 bitne operacijske kode. Ob izvajanju se ti 16 bitni ukazi razširijo v polne 32 bitne ukaze ARM ali pa se izvedejo direktno s pomočjo namenske dekodirne enote Thumb. Generirana programska koda Thumb potrebuje 40 % več ukazov kot

ekvivalentna 32 bitna ARM, vendar porabi tipično 30 % manj pomnilniškega prostora in je 40 % bolj počasna, zato se običajno uporablja za nekritične operacije, ki zahtevajo majhno porabo pomnilnika[7].

4.6.2 Razširitve DSP

Sistemi, temelječi na jedru ARM, so tipično uporabljali poseben DSP koprocesor za obdelavo signalov, vendar je v določenih primerih, na primer pri dekodiranju MP3, dobro imeti na razpolago operacije in ukaze DSP podprte direktno v jedru. Ta podpora je bila prvič dodana v peti generaciji arhitekture ARM, ki je omogočala 16 bitno množenje s seštevanjem (angl. multiply-accumulate) v eni urini periodi.

V šesti in sedmi generaciji se je ta podpora DSP razvila v čiste ukaze SIMD, ki omogočajo istočasno izvajanje dveh 16 bitnih ali štirih 8 bitnih aritmetičnih operacij. Naslednjo stopničko v razvoju pa predstavlja nabor ukazov NEON, ki omogoča izvajanje SIMD operacij na 128 bitnem vektorju. Za razliko od prejšnjih verzij, so lahko podatkovni tipi predznačena ali nepredznačena 8, 16, 32 ali 64 bitna števila s plavajočo vejico z enojno natančnostjo [7].

Poglavje 5 Platforma LUX9

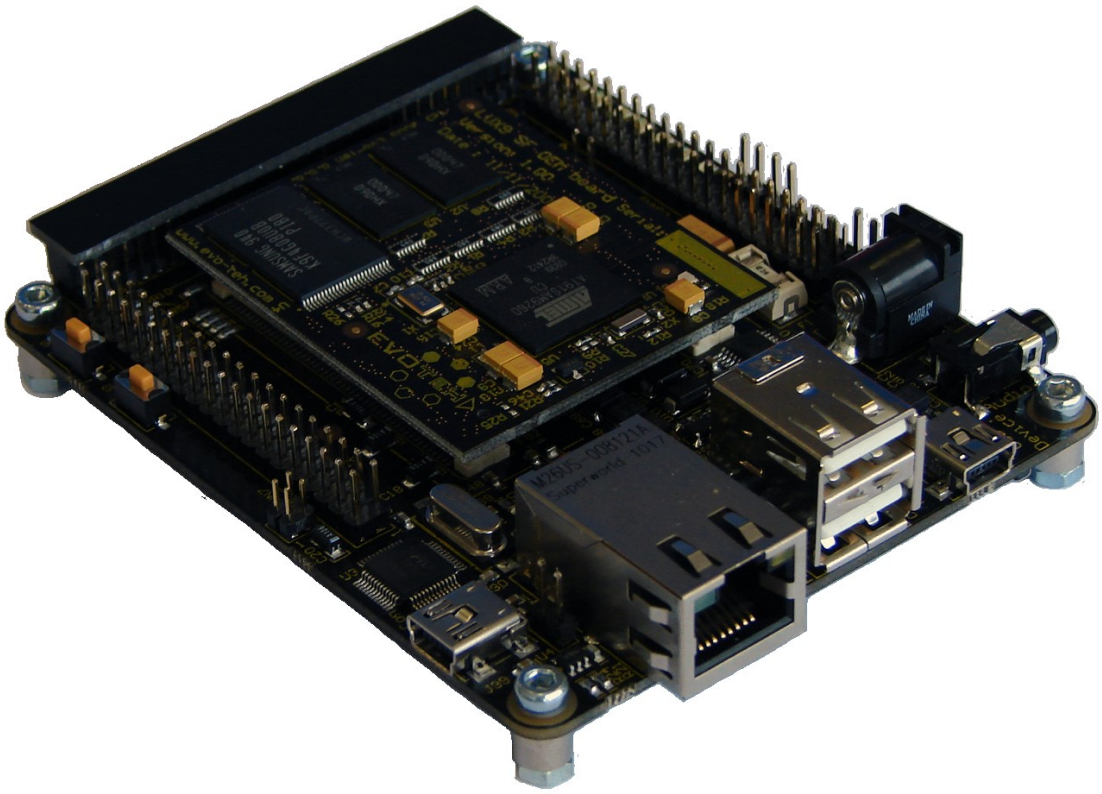
Platformo LUX9 sestavlja družina več modulov, pripadajočih osnovnih plošč ter evalvacijska razvojna ploščica. Pri načrtovanju modulov je bil uporabljen koncept sistem na modulu (angl. system-on-module, SoM). Moduli SoM predstavljajo kompletne vgrajene računalnike, ki so zasnovani na samostojnem tiskanem vezju, na katerem se nahajajo mikroprocesor in zunanji pomnilnik ter tako skupaj tvorijo funkcionalno delujoč računalnik. Ker moduli ne vsebujejo standardnih priključkov za priklop vhodno izhodnih naprav potrebujejo nosilno osnovno ploščo za delovanje.

Za družino produktov LUX9 je bil kot osnova vgrajenega sistema izbran mikroprocesor podjetja Atmel, iz družine 32 bitnih procesorjev ARM. Atmel deli svoje na jedrih ARM temelječe procesorje na več poddružin. Moduli LUX9 uporabljajo mikroprocesorje iz dveh sorodnih produktnih linij in sicer SAM9G ter SAM926x. Obe liniji mikroprocesorjev SoC sta načrtovani okrog jedra ARM926EJ-S z različno velikostjo predpomnilnikov, različno frekvenco delovanja ter različnimi vhodno izhodnimi bloki na nivoju procesorja. Dobro poznavanje delovanja jedra ARM926EJ-S, opisanega v poglavju 4, je osnova in nujno potrebno pri razvoju sistemske in testne programske opreme za ta mikroprocesor in posledično celotno platformo LUX9.

Razvojna ploščica LUX-EDK9 in modul LUX-SF9 sta zasnovana na mikroprocesorju AT91SAM9260, modul LUX-SFX9 na mikroprocesorju AT91SAM9263, vsi iz družine SAM926x, modul LUX-SF9G pa na mikroprocesorju AT91SAM9G20 iz družine SAM9G. V nadaljevanju se bomo osredotočili na naprave z mikroprocesorjema AT91SAM9260 in AT91SAMG20, ker so si poleg jedra ARM926 podobne tudi pri podprtih vhodno izhodnih napravah [16].

5.1 Razvojni komplet LUX9

Razvojni komplet LUX9 je namenjen spoznavanju funkcionalnosti in razvoju programske opreme za module SoM iz družine LUX-SF9. Komplet vsebuje enega ali več kompatibilnih modulov ter pripadajočo osnovno ploščo Starport. Na sliki 10 je prikazana osnovna plošča Starport s pripadajočim modulom LUX-SF9.



Slika 10: Osnovna plošča Starport z modulom LUX-SF9.

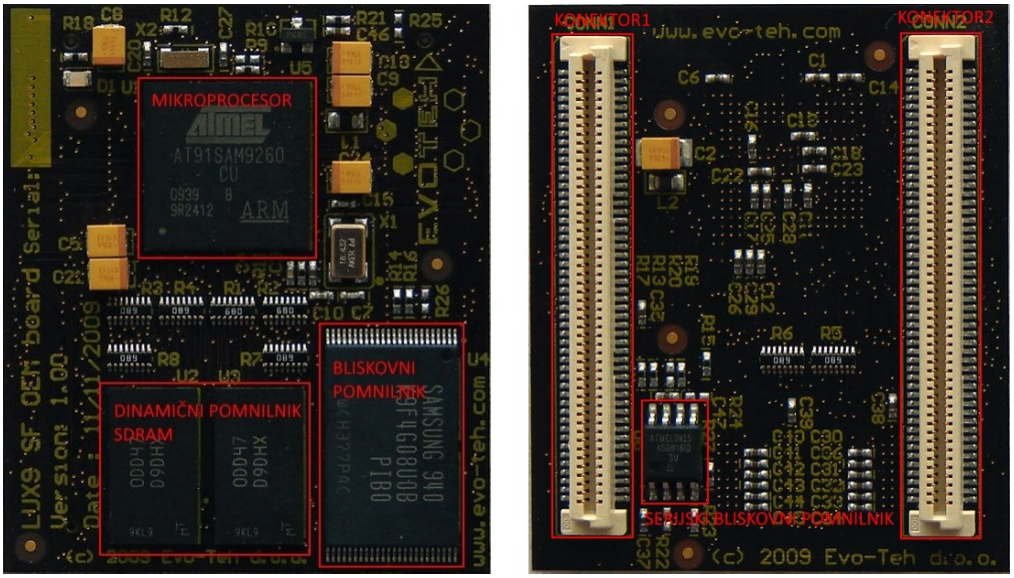
Osnovna plošča je zasnovana tako, da omogoča uporabo več različnih modulov LUX-SF9 in ostalih modulov temelječih na kompatibilnih mikroprocesorjih. Modulom zagotavlja potrebno napajanje, dostop do vhodno izhodnih enot preko standardnih priključkov (USB, Ethernet, UART) ter dveh razširitvenih letvic, na katere so speljane ostale proste nožice mikroprocesorja na modulu. Na ta način je omogočen dostop do vseh perifernih enot mikroprocesorja za potrebe razvoja ali testiranja.

Uporaba modulov končnemu uporabniku omogoča, da se v času snovanja novega produkta posveti razvoju nosilnega tiskanega vezja s perifernimi enotami in priključki, ki jih zahteva projekt. Prihranjena mu je časovna in tehnološka zahtevnost razvoja in testiranja delujočega računalniškega sistema na večplastnem tiskanem vezju. Poleg tega lahko, če se pokaže v prihodnosti potreba, počasnejši modul samo preprosto nadomesti s kompatibilnim hitrejšim.

5.1.1 Modula LUX-SF9 in LUX-SF9G

Modula LUX-SF9 in LUX-SF9G sta si vizualno in tudi funkcionalno zelo podobna, saj uporabljata med seboj na nivoju nožic združljiva procesorja iz dveh različnih poddružin in sicer

Atmel SAM926x ter SAM9G. Oba modula uporabljata za osnovo 6 slojno tiskano vezje, na katerem se nahajajo osnovne komponente sistema na modulu: mikroprocesor, dinamični pomnilnik SDRAM, bliskovni pomnilnik NAND, serijski bliskovni pomnilnik in dva konektorja. Na sliki 11 je predstavljen modul LUX-SF9 z označenimi osnovnimi komponentami.



Slika 11: Modul LUX-SF9 od zgoraj levo in od spodaj desno.

Specifikacije obeh modulov LUX-SF9 in LUX-SF9G so opisane v tabeli 5:

Tabela 5: Specifikacija modulov LUX-SF9 in LUX-SF9G.

Naziv	**LUX-SF9**	**LUX-SF9G**
Mikroprocesor	Atmel AT91SAM9260-QU	Atmel AT91SAM9G20-CU
Frekvenca CPE	200 MHz	400 MHz
Pomnilnik SDRAM	64 MB (FSB 100 MHz)	64 MB (FSB 133 MHz)
Bliskovni (NAND Flash) pomnilnik	512 MB	512 MB
Serijski (DATA Flash) pomnilnik	4 MB	4 MB
Ostalo	2 konektorja	2 konektorja

5.1.2 Mikroprocesor Atmel AT91SAM9G20

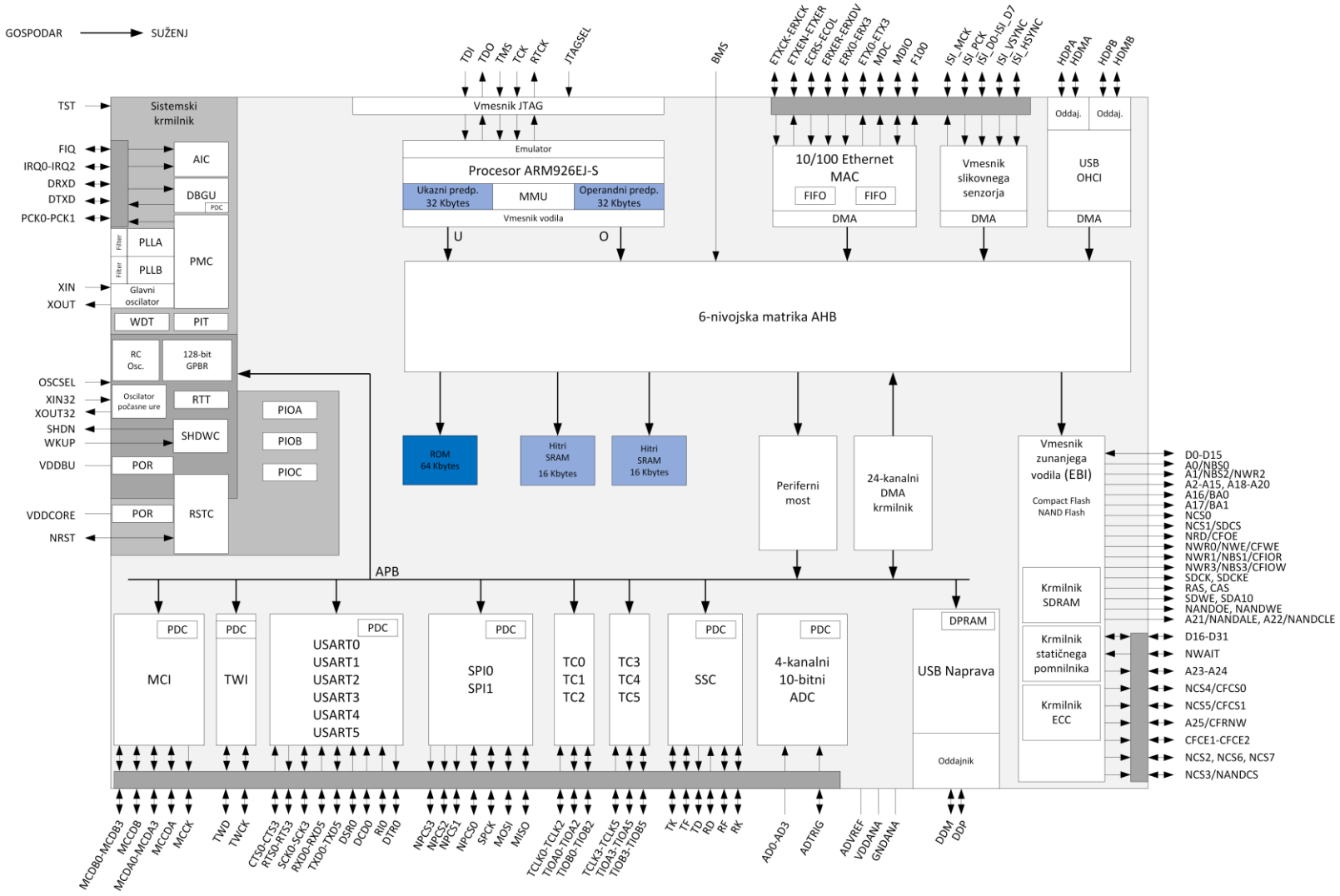
Mikroprocesor Atmel AT91SAM9G20 je zasnovan okoli jedra ARM926EJ-S, hitrih notranjih pomnilnikov ROM in RAM, ter večjega števila perifernih enot. Poleg osnovnih perifernih enot kot so USART, SPI, TWI, ADC in krmilnik za kartice MMC vključuje procesor še mrežni krmilnik Ethernet in krmilnik za USB naprave z integriranimi oddajniki.

Vse te komponente komunicirajo med sabo preko 6 nivojske matrike vodil, kar omogoča maksimalno notranjo pasovno širino šestih 32 bitnih vodil pri hitrosti 133 MHz. Zunanji pomnilniški vmesnik pa omogoča priklop kar največjega števila različnih tipov zunanjih pomnilnikov, kot so statični, dinamični in bliskovni pomnilnik.

Samo jedro ARM926EJ-S, ki lahko deluje pri maksimalni frekvenci 400 MHz in temelji na arhitekturi ter ukazih ARMv5TEJ, vključuje izboljšan 16 x 32 bitni množilnik ter 16 bitni digitalni procesor signalov z enoto MMU, potrebno za poganjanje sodobnih operacijskih sistemov kot je Linux. Jedru ARM je na voljo 32 KB podatkovnega in 32 KB ukaznega predpomnilnika.

Mikroprocesor vsebuje še dva interna statična pomnilnika SRAM velikosti po 16 KB z dostopnim časom enega procesorskega cikla ter interni bralni pomnilnik ROM velikosti 64 KB, v katerem je shranjena koda potrebna za zagon procesorja in inicializacijo zunanjih pomnilniških sistemov. Trije različni krmilniki zunanjih pomnilnikov, statični, dinamični in krmilnik ECC upravljajo z zunanjim 32 bitnim vodilom za priklop zunanjih pomnilnikov. Vodilo omogoča priklop zunanjega statičnega, dinamičnega SDRAM in nizkonapetostnega pomnilnika Mobile SDRAM ter bliskovnega pomnilnika SLC NAND Flash s strojnim odpravljanjem napak ECC.

Integrirani mrežni krmilnik Ethernet MAC je združljiv s standardom IEEE 802.3 in omogoča hitrost prenosa podatkov 10/100 Mbit z možnostjo priklopa na fizični nivo s pomočjo vmesnikov MII ali RMII. Oba krmilnika za naprave USB in gostitelja USB omogočata polno hitrost po standardu USB v2.0 in sicer 12 Mbit. Poleg serijskega, TWI, UART in MMC krmilnika je na voljo še 4 kanalni 10 bitni analogno digitalni pretvornik. Celotna blok shema z vsemi enotami mikroprocesorja je podrobneje predstavljena na sliki 12 [17].



Slika 12: Bločna shema mikrokrmilnika SAM9260.

5.1.3 Zunanji pomnilniki

Na zunanje podatkovno vodilo mikroprocesorja AT91SAM9G20 sta priključena dva tipa pomnilniških čipov:

- Zunanji dinamični pomnilnik SDRAM, ki ga sestavljata dva BGA čipa Micron MT48LC16M16 (velikost 256 Mbit = 4 Mbit x 16 x 4 banke), priklopljena na 32 bitno podatkovno vodilo.
- Zunanji NAND FLASH pomnilnik Samsung K9F4G08U0A-P (velikost 4 Gbit = 512 Mbyte) in 8 bitnim podatkovnim vodilom.

5.1.4 Mrežni vmesnik Ethernet

Mikroprocesor na modulu vsebuje integriran krmilnik Ethernet MAC s hitrostjo prenosa podatkov 10/100 Mbit/s in z vmesnikom za zunanji analogno digitalni pretvornik PHY. Krmilnik je s procesorjem povezan preko izoliranega podatkovnega vodila z namenskim pomnilnikom za hitro izmenjavo podatkov.

Na ploščici je uporabljen Ethernet PHY DP83848 podjetja National, priključen na mikroprocesor preko vmesnika RMII. Za generiranje ure, ki jo predpisuje vmesnik RMII, je uporabljen kristal s frekvenco 50 MHz.

5.1.5 Napajanje

Razvojna osnovna plošča Starport je načrtovana tako, da se lahko napaja iz dveh neodvisnih virov in sicer vmesnika USB ali pa enosmernega napajalnika s pozitivno napetostjo med 9 in 15 V. Mikroprocesor ter ostali čipi ter periferne naprave potrebujejo za svoje delovanje različne napajalne napetosti, ki jih zagotavljata stikalni pretvornik DC/DC LM2717-ADJ (3,3 in 5 V) ter linearni pretvornik DC/DC LP3878-ADJ (1,8 V) podjetja Texas Instruments.

Poglavje 6 Programska oprema na platformi LUX9

Vsak računalnik ali druga vgrajena strojna oprema potrebuje programsko opremo, ki jo poganja, da lahko služi svojemu namenu. Zato je pomembno, da proizvajalci strojne opreme, v našem primeru sistema na modulu, zagotovijo kvalitetno podporo na nivoju operacijskega sistema. V zadnjem času se je kot prevladujoči operacijski sistem na področju vgrajenih sistemov uveljavil Linux. Na srečo je bilo jedro Linux načrtovano tako, da omogoča relativno enostaven prenos na druge še ne podprte sisteme, vendar pa je potrebno zelo dobro poznavanje strojne opreme, v našem premeru arhitekture ARM9 (poglavje 4), mikroprocesorja Atmel AT91SAM9G20 in osnovne plošče Starport, oba opisana v poglavju 5. Dobro poznavanje strojne opreme, je torej pogoj za kvalitetno načrtovanje in prenos operacijske programske opreme na novo platformo.

6.1 Zakaj Linux?

Uporaba operacijskega sistema Linux se je v zadnjih letih razširila iz klasično prevladujočega okolja strežniških sistemov in superračunalnikov v ekosistem vgrajenih naprav. Primeri uporabe v vgrajenih sistemih so predvsem mobilni telefoni (Android, Tizen, Sailfish, Ubuntu), večpredstavni predvajalniki, digitalne kamere in omrežna oprema kot so usmerjevalniki in stikala.

Zaradi številnih ekonomskih in tehničnih prednosti, glede na konkurenco, lahko v zadnjem obdobju opazimo zelo veliko rast uporabe tega operacijskega sistema v vgrajenih napravah. Ta trend se pozna na vseh področjih od industrije do zabavne elektronike za široko uporabo. Poglavitni razlogi za takšno rast uporabe sistema Linux so predvsem:

- podpora velikemu številu različnih strojnih naprav in arhitektur (ARM, MIPS, x86, amd64, PowerPC, SPARC),
- podpora velikemu številu aplikacij in mrežnih protokolov,
- razširljivost vse od malih naprav in sistemov na čipu do stikal in usmerjevalnikov ter industrijskih naprav,
- brez licenčnin, kot pri standardnih lastniških rešitvah,

- kvalitetna in prosto dostopna razvojna orodja (prevajalniki, razhroščevalniki, integrirana razvojna orodja),
- veliko število aktivnih razvijalcev, ki zagotavljajo hitro podporo za nove strojne arhitekture, platforme in naprave,
- podpora s strani skoraj vseh največjih in najpomembnejših proizvajalcev integriranih vezij (procesorjev, mikrokrmilnikov).

Poleg vseh naštetih, je eden izmed glavnih faktorjev, ki je botroval tako množični uporabi sistema Linux v vgrajenih napravah dejstvo, da je izvorna koda prosto dostopna. Jedro Linux je licencirano in dostopno pod licenčnimi pogoji GNU GPL[1]. Za programsko opremo licencirano pod temi pogoji je značilno, da njeno izvorno kodo lahko uporabniki svobodno uporabljajo, študirajo in spreminjajo. Kar pa ne pomeni, da je zastonj, ampak prosta za uporabo. Vsak inženir razume, da nastanejo določeni stroški že z uporabo programske opreme v napravah. Ti stroški so povezani s pridobitvijo, integracijo, spremembami in podporo te izvorne kode za vsako razvito napravo skozi njen celoten življenjski cikel.

Zgoraj našteti razlogi so privedli do odločitve, da je potrebno za družino modulov in razvojnih ploščic LUX9 zagotoviti popolno podporo operacijskemu sistemu Linux. Uporaba poznanega in odprtega operacijskega sistema zagotavlja tudi kar se da širok krog uporabnikov in razvijalcev, ki bodo lahko uporabljali module v svojih končnih produktih [4, 15].

6.2 Umestitev sistema Linux v vgrajeno napravo

Razumevanje osnovnih konceptov pri razvoju sistemske programske opreme za vgrajene sisteme predstavlja začetnikom velikokrat oviro, zato je nujno potrebno predhodno poznavanje celotne slike delovanja sistema. Pri prilagajanju operacijskega sistema Linux za specifično vgrajeno napravo se je potrebno seznaniti s koncepti in komponentami, ter njihovo umestitvijo v verigo dogodkov, ki se izvede ob priklopu napajanja naprave pa do zagona operacijskega sistema.

6.2.1 Zaganjalnik

Tipično je v vgrajenih napravah zaganjalnik (če prevzamemo, da to ni naprava, ki temelji na platformi x86, kjer to nalogo opravi BIOS ali UEFI) prvi program, ki prevzame nadzor nad procesorjem. To je skupek sistemskih konfiguracijskih funkcij in procedur, ki so tesno povezane in odvisne od nizko ležeče strojne opreme. Njihova glavna naloga je inicializacija

[1] Celoten tekst licence je dosegljiv na http://www.gnu.org/licenses/gpl.html.

strojne opreme, še posebej pomnilniškega podsistema in kopiranje ter zagon jedra operacijskega sistema iz bliskovnega pomnilnika ali pomnilniških kartic. Pomemben del zagotavljanja podpore sistemu Linux je razvoj ustreznega zaganjalnika za napravo [4]. Na srečo obstaja več odprto kodnih projektov, ki se jih lahko uporabi in prilagodi za te potrebe.

Pomembne naloge, ki jih mora zaganjalnik opraviti so:

- Inicializacija kritičnih strojnih komponent, kot je krmilnik SD/DDR RAM, prekinitveni krmilnik, vhodno izhodne naprave in grafični krmilnik.
- Inicializacija sistemskega pomnilnika v povezavi s predajo nadzora jedru operacijskega sistema.
- Zagotoviti mehanizem za lociranje in nalaganje slike operacijskega sistema.
- Naložiti sliko in predati nadzor operacijskemu sistemu, ter mu pripraviti potrebne zagonske parametre, kot so informacija o količini fizičnega pomnilnika in hitrost serijskih vrat.

V našem primeru smo morali zaradi omejitev, ki jih postavlja arhitektura sistema na čipu in s tem vgrajen program BootROM, ki se nahaja na pomnilniku ROM integriranem v procesorju, uporabiti dodaten prvo stopenjski zaganjalnik, ki naloži drugo stopenjskega, v našem primeru v svetu vgrajenih naprav razširjeni U-Boot. Ko ta prevzame nadzor nad procesorjem ustrezno inicializira serijsko povezavo UART, bliskovni krmilnik in mrežni krmilnik. Naloga inicializacije krmilnika SDRAM je s tem prestavljena na prvo stopenjski zaganjalnik.

6.2.2 Nalaganje in zagon jedra

Ko zaganjalnik inicializira potrebno strojno opremo, je njegova naloga, da naloži v pomnilnik sliko jedra, mu poda potrebne parametre in ga zažene. To lahko naredi na več načinov, odvisno od arhitekture strojne opreme. Slika jedra se lahko nahaja na različnih medijih. Zaganjalnik jo lahko skopira preko mreže s pomočjo protokola TFTP ali pa bliskovnega pomnilnika na ustrezno pomnilniško lokacijo, ki jo določa arhitektura. Zaganjalnik mora jedru, preden ga zažene, še na določeni lokaciji pripraviti strukturo s parametri, specifičnimi za napravo. To je tudi njegovo zadnje dejanje, ker potem preneha obstajati. Jedro Linux prevzame popoln nadzor nad strojno opremo in pomnilnikom, tudi tistimi deli pomnilnika, ki jih je prej uporabljal zaganjalnik. Njegov zagon je mogoč samo ob ponovnem zagonu naprave.

Ob zagonu je ena od prvih nalog jedra Linux konfiguracija enote za upravljanje s pomnilnikom (MMU) in podatkovnih struktur za njeno podporo. Sledi vklop preslikovalnika naslovov, ta omogoča jedru operacijskega sistema visok nivo upravljanja in nadzora nad naslovnim

prostorom, ki ga dodeljuje procesom. Uporaba navideznega pomnilnika omogoča jedru bolj učinkovito upravljanje nad fizičnim pomnilnikom in uveljavljanje pravic dostopa vsakemu procesu in opravilu posebej, da ne bi prišlo do nepredvidenega dostopa med njimi.

Ko je ta korak končan, jedro teče v svojem navideznem pomnilniškem prostoru, ki se v zadnjih verzijah jedra začne na pomnilniškem naslovu 0xc0000000. Za večino arhitektur je to nastavljiv parameter, vendar se ta možnost redko uporablja.

Za jedro Linux sta značilna dva konteksta delovanja glede na okolje v katerem se izvajajo programi oziroma ukazi. Niti, ki se izvajajo znotraj jedra, pravimo da delujejo v kontekstu jedra (angl. kernel mode), aplikacije pa v uporabniškem kontekstu (angl. user mode). Proces v uporabniškem kontekstu lahko dostopa do uporabniškega pomnilnika, ki mu je bil dodeljen in mora uporabiti programske klice jedra, če želi dostopati do privilegiranih sredstev, kot so datoteke in vhodno/izhodne naprave.

Preden jedro zaključi svoje inicializacijo in se na terminalu običajno prikaže prijavna (angl. login) vrstica mora priključiti korenski (angl. root) datotečni sistem. Za Linux je značilno, da za svoje delovanje potrebuje korenski datotečni sistem, kar za druge vgrajene sisteme ni nujno potrebno.

Ko jedro Linux konča z interno inicializacijo in priklopom korenskega datotečnega sistema, je njegova privzeta naloga, da zažene program `init` v uporabniškem kontekstu. V tem načinu delovanja ima delujoč program omejen dostop do sistema in mora uporabiti sistemske klice jedra za dostop do naprav in datotek. Ti uporabniški procesi tečejo v navideznem pomnilniškem prostoru, ki ga jedro izbere naključno.

6.2.3 Vgrajena distribucija Linux

Po izvedenem začetnem zagonskem delu, jedro Linux pričakuje, da bo našlo in priklopilo korenski datotečni sistem. Priklopu korenskega datotečnega sistema sledi zagon skript ali nadzornega programa, ki zažene večje število programov in pripomočkov, ki so potrebni za normalno delovanje sistema. Ti programi se velikokrat sklicujejo na druge, da opravijo specifične naloge, kot je zagon ukazne vrstice, nastavitev omrežnih naprav in zagon uporabniških aplikacij. Vsak od teh programov ima specifične zahteve (angl. dependencies), ki jih morajo zagotoviti druge komponente sistema. Večina programov zahteva prisotnost ene ali več sistemskih knjižnic. Drugi programi spet potrebujejo konfiguracijske ali dnevniške datoteke. Če povzamemo, tudi najmanjši vgrajeni sistemi Linux vsebujejo in potrebujejo večje število datotek porazdeljenih v določeno hierarhično strukturo map v korenskem datotečnem sistemu.

Datoteke, ki spadajo eni aplikaciji ali programu, se združujejo v pakete zaradi lažjega upravljanja in nadgrajevanja sistema. Več teh paketov pa predstavlja celoto, ki jo poznamo kot distribucijo operacijskega sistema Linux. Distribucijo sestavljajo paketi uporabnih programov, knjižnic, orodij, pripomočkov in dokumentacije.

6.2.4 Navzkrižno razvojno okolje

Da bi lahko razvijali aplikacije in gonilnike za vgrajene sisteme potrebujemo ustrezna orodja (prevajalnik, povezovalnik, razne pripomočke), ki generirajo binarne datoteke v ustreznem formatu, da jih lahko potem poženemo na ciljni napravi. Če napišemo kratek program na osebnem računalniku in ga s prevajalnikom prevedemo dobimo ustrezno binarno obliko, ki jo lahko zaženemo samo na tej arhitekturi. Temu pravimo prevajanje za domačo (angl. native) napravo. Podobno velja, če imamo na voljo razvojna orodja na naši vgrajeni napravi in program lahko prevedemo direktno na njej. To je lahko recimo dober stresni test pravilnega delovanja naprave, če na njej večkrat prevedemo jedro Linux. Vendar to velikokrat ni smotrno, ker je ciljna naprava, v našem primeru družina LUX9, procesorsko in s količino pomnilnika precej podhranjena. Takšno početje bi zahtevalo veliko časa, ki pa je v razvoju zelo dragocen. Prevajanje jedra bi tako trajalo kar nekaj ur, zato se poslužimo tako imenovanega navzkrižnega prevajanja (angl. cross compiling), kjer na eni arhitekturi v našem primeru x86 prevedemo s posebej pripravljenimi orodji programe za drugo ciljno arhitekturo, pri nas ARM.

Priprava takšnega navzkrižnega razvojnega okolja je zahtevna, predvsem pa vsebuje kopico skritih pasti. Ko od prevajalnika zahtevamo, da prevede izvorno kodo programa, v večini primerov avtomatsko najde datoteke in knjižnice, ki jih mora vključiti, da se bo prevajanje uspešno izvedlo. To je možno, ker ima nastavljene privzete poti do zbirke vključitvenih datotek in knjižnic. Podobno deluje povezovalnik z vgrajeno privzeto potjo do knjižnice C, ko išče referenco do zunanjega simbola. Zbirke orodij za navzkrižno prevajanje (angl. toolchain) morajo tako vsebovati pravilne nastavitve do povezanih programov in knjižnic [13].

6.3 Zaganjalnik za družino mikroprocesorjev Atmel SAM926x in SAM9G

Zaganjalnik je kritična komponenta vsakega vgrajenega sistema in predstavlja temelj za zagon primarne sistemske programske opreme, v našem primeru jedra operacijskega sistema Linux. V tem poglavju se bomo osredotočili na vlogo zaganjalnika in pokazali kako moramo pripraviti okolje za uspešen zagon jedra Linux.

6.3.1 Vloga zaganjalnika

Ko priključimo napajanje sistema, je potrebno inicializirati osnovne komponente strojne opreme, preden lahko poženemo še tako preprost program. Vsak procesor ima predvidene določene akcije, ki se morajo izvesti, ko spustimo signal RESET. To vključuje nalaganje inicializacijske kode iz neke vrste pomnilnika (običajno notranji pomnilnik ROM). Ta zgodnja inicializacijska koda je del zaganjalnika in je odgovorna, da vlije življenje v procesor in povezane strojne komponente.

Večina procesorjev ima definiran privzet naslov s katerega pobere prve ukaze, ki jih izvede takoj po zagonu in s tem omogoči prevzem nadzora delovanja programski opremi. V primeru družine procesorjev Atmel SAM926x in SAM9G, ki temeljijo na arhitekturi ARM9, je to naslov 0x0.

Sistem se vedno zažene na naslovu 0x0. Da bi lahko zagotovili več možnosti pri izbiri načina zagona sistema, se lahko razporeditev pomnilnika nastavlja z dvema parametroma, konfiguracijskim bitom REMAP in signalom BMS preko nožice na procesorju.

Konfiguracijski bit REMAP omogoča programerju, ko se je sistem že zbudil, da preslika naslov 0x0 na prvo lokacijo internega pomnilnika SRAM0 in s skokom na prvo lokacijo prične izvajati kodo.

Signal BMS omogoča izbiro zagona sistema iz internega pomnilnika ROM ali pa zunanjega pomnilnika preko krmilnika SRAM in signala CS0. Če je signal BMS = 1 se iz internega pomnilnika ROM zažene Atmelov zagonski program BootROM, ki poskrbi za minimalno inicializacijo procesorja in nalaganje naslednjega programa iz drugih pomnilnikov. V primeru signala BMS = 0, moramo sami zagotoviti ustrezni inicializacijski program na zunanjem pomnilniku. Vse možne kombinacije vrednosti signala BMS in konfiguracijskega bita REMAP so navedene v tabeli 6.

Tabela 6: Razporeditev pomnilnika.

Address	**REMAP = 0**		**REMAP = 1**
	BMS = 1	BMS = 0	
0x0000 0000	ROM	EBI_NCS0	SRAM0 16K
0x0010 0000	ROM		
0x0020 0000	SRAM0 16K		
0x0030 0000	SRAM1 16K		
0x0050 0000	USB Host User Interface		

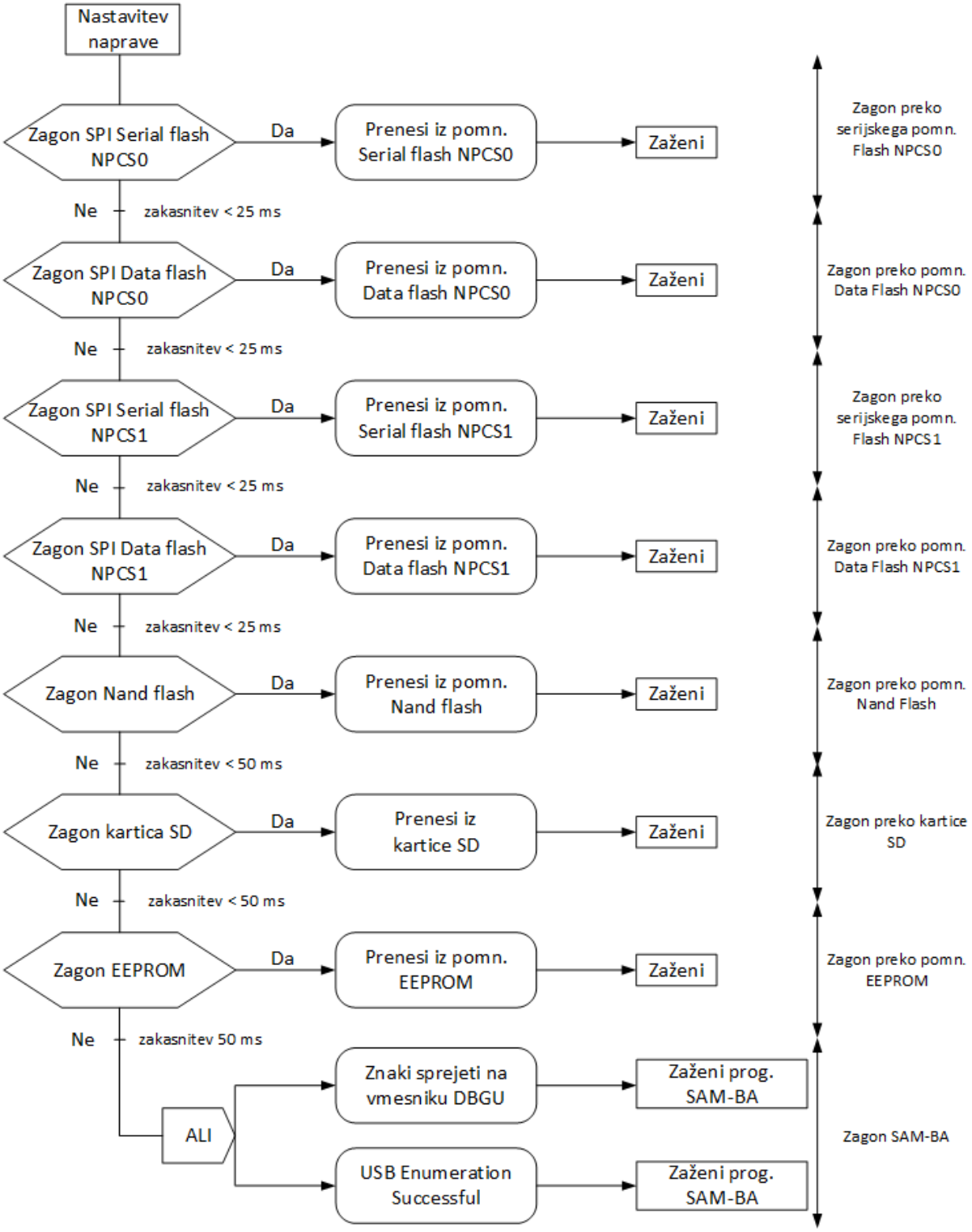
Vse naprave LUX9 uporabljajo kombinacijo signala BMS = 1. S tem je omogočen zagon programa BootROM iz internega pomnilnika ROM [18].

6.3.2 Atmel BootROM

Atmel BootROM je prvi program, ki se zažene v sklopu procesa zagona procesorja na platformi LUX9, ob predpostavki, da je postavljen signal BMS = 1. Njegova naloga je osnovna inicializacija procesorja, ki mu sledi iskanje veljavnega prvo stopenjskega zaganjalnika na podprtih zunanjih pomnilnikih v skladu z določenim vrstnim redom. Naloge, ki jih BootROM opravi so:

- nastavitev sklada za prehod v nadzorniški (angl. supervisor) način delovanja jedra ARM;
- inicializacija in zagon s pomočjo počasne ure (RC na čipu ali 32,768 Hz zunanji kristal);
- inicializacija sistemske konzole (BGU) in avtomatska detekcija hitrosti prenosa podatkov preko serijskih vrat;
- kopiranje in zagon aplikacije iz zunanjega medija v interni pomnilnik SRAM;
- avtomatska detekcija veljavnosti aplikacije (v našem primeru prvo stopenjskega zaganjalnika);
- zagon protokola SAM-BA, če ni bilo na voljo veljavne aplikacije na zunanjem trajnem pomnilniku (napačen zaganjalnik ali neprogramirana naprava).

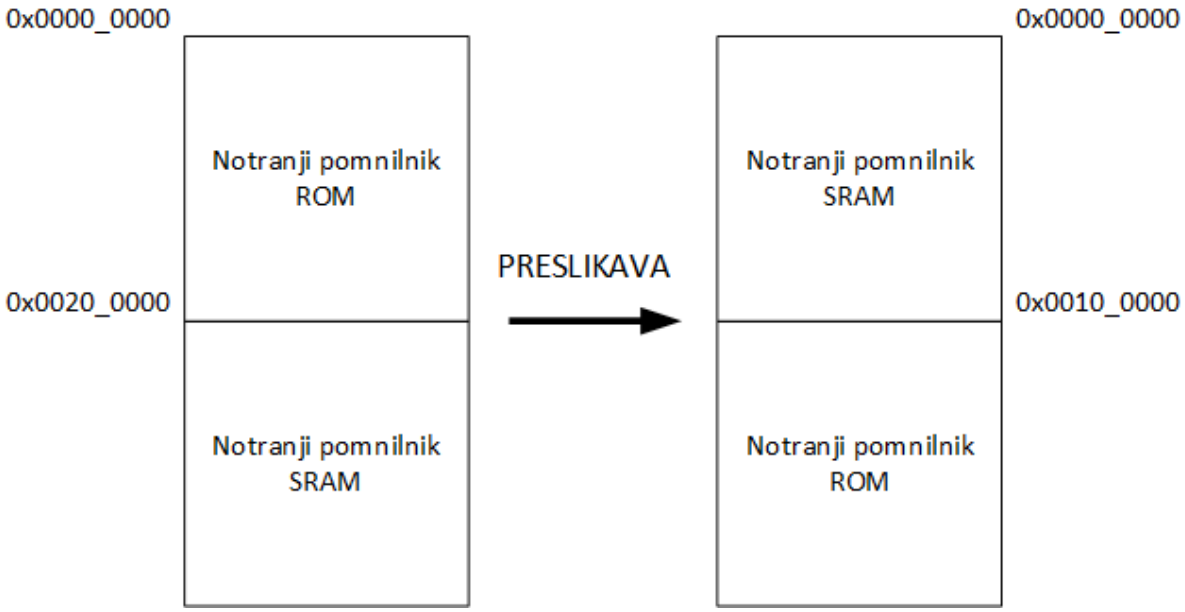
Na sliki 13 je podan diagram poteka zagona procesorja ob priklopu napajanja in zagon programa BootROM. Podane so posamezne stopnje in časovne zakasnitve pri preverjanju zunanjih pomnilnikov če vsebujejo veljavno verzijo zaganjalnika prve stopnje. Diagram je podan za procesor Atmel AT91SAM9G20, ki je novejši in podpira večje število različnih zunanjih pomnilnikov. Starejši procesorji na modulih družine LUX9 AT91SAM9260 in AT91SAM9263, podpirajo manjše število različnih kombinacij zunanjih pomnilnikov zaradi manjšega pomnilnika ROM in starejše verzije integriranega programa BootROM. Kljub temu pa je logika delovanja povsem enaka in je navedena v dokumentaciji omenjenih mikroprocesorjev [17, 18].



Slika 13: Diagram poteka izvajanja programa BootROM.

Naloga programa BootROM je, da poišče veljaven program za zagon procesorja. To naredi tako, da pregleda prvih 28 bajtov prekopirane binarne kode in ugotovi prisotnost ustreznih

prekinitvenih vektorjev ARM, ki morajo vsebovati ukaze za vejitev ali pa nalaganje programskega števca z relativnim naslavljanjem. Šesti vektor mora na odmiku 0x14 vsebovati velikost slike programa (zaganjalnika), ki ga je potrebno skopirati v interni pomnilnik SRAM. Velikost zaganjalnika je omejena z velikostjo pomnilnika SRAM, v primeru mikroprocesorja AT91SAM9260 je to 4 KB, kar predstavlja kar precejšnjo omejitev pri načrtovanju in programiranju prvostopenjskega zaganjalnika. Po kopiranju programa v pomnilnik SRAM sledi preslikava naslova 0x0 na prvo pomnilniško lokacijo pomnilnika SRAM in nastavitev programskega števca PC na 0 ter s tem zagon programa. Na sliki 14 je prikazano kako se interni pomnilnik SRAM preslika na novo pomnilniško lokacijo, čemur po navadi sledi zagon programa.



Slika 14: Prerazporeditev dostopa do pomnilnika.

Če BootROM ne najde veljavnega programa na definiranih lokacijah se nazadnje zažene protokol SAM-BA, ki preko vrat USB omogoča dostop do pomnilnika procesorja, direktno kopiranje programov v SRAM, dostop do registrov in zaganjanje programov. S pomočjo tega protokola se izvede tudi produkcijsko testiranje vseh sistemov LUX9 in začetni prenos sistemske programske opreme na ploščico. Tabela 7 prikazuje nabor ukazov, ki jih podpira protokol SAM-BA [18].

Tabela 7: Ukazi protokola SAM-BA.

Ukaz	Opis	Argument ukaza	Primer
O	zapiši zlog	Address, Value#	O200001,CA#
o	preberi zlog	Address,#	o200001,#
H	zapiši pol besede	Address, Value#	H200002,CAFE#
h	preberi pol besede	Address,#	h200002,#
W	zapiši besedo	Address, Value#	W200000,CAFEDECA#
w	preberi besedo	Address,#	w200000,#
S	pošlji datoteko	Address,#	S200000,#
R	sprejmi datoteko	Address, NbOfBytes#	R200000,1234#
G	zaženi	Address#	G200200#
V	prikaži verzijo	No argument	V#

6.3.3 Zaganjalnik AT91Bootstrap

AT91Bootstrap je prvo stopenjski zaganjalnik, ki ga BootROM iz zunanjega pomnilnika naloži v interni pomnilnik SRAM in zažene. Njegova naloga je, da inicializira sistemsko uro, nastavi vhodno izhodne povezave, ter zbudi in za delovanje pripravi pomnilnik SDRAM in potem naloži drugo stopenjski zaganjalnik, v našem primeru U-Boot.

Zaganjalnik AT91Bootstrap namenjen modulom LUX9 je bil razvit na osnovi demo projekta AT91Bootstrap podjetja Atmel in temelji na kompletu razvojnih knjižnic AT91LIB za programski jezik C družine procesorjev SAM9. Program je razvit modularno, tako da so vsi člani družine LUX9 in s tem tudi različni mikroprocesorji podprti znotraj istega projekta. Zaradi modularne zasnove je dodajanje podpore za nove procesorje in module preprosto.

Projekt je fizično razdeljen v dve podmapi `at91bootstrap` [20] in `at91lib` [19]. V mapi `at91bootrap` se nahaja datoteka `main.c`, ki predstavlja glavni program. Mapa `at91lib` vsebuje gonilnike za delo z bliskovnimi pomnilniki, iz katerih se lahko naloži drugo stopenjski zaganjalnik. Glavni program je zelo preprost in je sestavljen samo iz prototipov funkcij, ki so potrebne, da se pripravi okolje za prenos in zagon drugo stopenjskega zaganjalnika. Najbolj pomembni deli glavnega programa predstavljajo naloge, ki jih opravi prvo stopenjski zaganjalnik:

a. nastavitev GPIO

```
static const Pin pinsCustom[] = {PINS_CUSTOM};
static const Pin pinsLED[] = {PINS_LEDS};
PIO_Configure(pinsCustom, PIO_LISTSIZE(pinsCustom));
PIO_Configure(pinsLED, PIO_LISTSIZE(pinsLED));
```

b. inicializacija zunanjega pomnilnika

```
BOARD_ConfigureVddMemSel(VDDMEMSEL_3V3);
BOARD_ConfigureSdram(BOARD_SDRAM_BUSWIDTH);
```

c. kopiranje drugo stopenjskega zaganjalnika v pomnilnik (iz pomnilnika DataFlash)

```
BOOT_AT45_CopyBin(tabDesc, TDESC_LISTSIZE(tabDesc));
```

d. zagon prekopiranega programa (U-Boot)

```
GoToJumpAddress(tabDesc[0].dest, MACH_TYPE);
```

Za vsako razvojno ploščico in procesor je potrebno posebej definirati podporo v obliki funkcij, ki jih kliče glavni program. V mapi `at91lib/boards` se nahajajo datoteke s kodo za inicializacijo napisano v zbirniku `board_cstartup.S`. Osnovne nastavitve frekvence ure za procesor, potrebne periferne enote in krmilnik prekinitev so definirani v datoteki `board_lowlevel.c`, inicializacija krmilnika pomnilnika SDRAM pa v datoteki `board_memories.c`. Pri inicializaciji pomnilnika SDRAM je potrebno nastaviti pravilne parametre, ki se bodo vpisali v register krmilnika zunanjega pomnilnika in jih določa proizvajalec pomnilniških čipov. Slika 15 prikazuje strukturo potrebno za inicializacijo eksternega pomnilnika v funkciji *`BOARD_ConfigureSdram`* za procesor AT91SAM9260 na ploščici LUX-SF9.

```
// CFG Control Register
WRITE(AT91C_BASE_SDRAMC, SDRAMC_CR, AT91C_SDRAMC_NC_9
                                  | AT91C_SDRAMC_NR_13
                                  | AT91C_SDRAMC_CAS_2
                                  | AT91C_SDRAMC_NB_4_BANKS
                                  | sdrc_dbw
                                  | AT91C_SDRAMC_TWR_2
                                  | AT91C_SDRAMC_TRC_7
                                  | AT91C_SDRAMC_TRP_2
                                  | AT91C_SDRAMC_TRCD_2
                                  | AT91C_SDRAMC_TRAS_5
                                  | AT91C_SDRAMC_TXSR_8);
```

Slika 15: Struktura za inicializacijo pomnilnika SDRAM.

Takšna struktura projekta nam omogoča preprosto dodajanje podpore za novejše čipe. Najprej je potrebno posodobiti knjižnico z gonilniki, dodati ustrezne funkcije ter napisati kodo za inicializacijo procesorja na najnižjem nivoju. S pomočjo nastavitvene datoteke `Makefile` lahko

nato prevedemo in povežemo program, ob tem pa navedemo ustrezne možnosti, kot so verzija procesorja, ploščica in nastavitve za pomnilnik. Na sliki 16 je prikazan primer kako prevesti zaganjalnik AT91Bootloader za mikroprocesor AT91SAM9260 na modulu LUX-SF9, ki bo omogočal prenos in zagon programa preko serijskega bliskovnega pomnilnika.

```
make CHIP=at91sam9260 BOARD=lux-sf9 ORIGIN=dataflash SLOT=SLOT_B DESTINATION=sdram
BIN_SIZE=0x35000 FROM_ADDR=0x8400 DEST_ADDR=0x23F00000 OP_BOOTSTRAP=on
STR_DESCR=\\\"u-boot\\\" TRACE_LEVEL=0 clean all
```

Slika 16: Primer ukaza za prevajanje zaganjalnika.

Velika pomanjkljivost procesorja AT91SAM9260 je, da ima samo 4 KB internega pomnilnika SRAM in moramo biti zelo preudarni pri pisanju kode in si pomagati tudi z ne najbolj preglednim ampak učinkovitim načinom programiranja. Da je velikost programa ostala pod zahtevano mejo 4KB, smo ga prevedli z uporabo nabora ukazov Thumb.

6.3.4 U-Boot

Na trgu lahko najdemo veliko različnih odprto kodnih in komercialnih zaganjalnikov (angl. bootloader), ki se uporabljajo v različnih vgrajenih sistemih. Za njih so značilne določene skupne lastnosti, predvsem da vsi omogočajo nalaganje in zagon drugega programa, največkrat operacijskega sistema. Glavni kanal za interakcijo z uporabnikom je serijski vmesnik, uporaba mrežnih protokolov in podpora Ethernetu pa je redkost zaradi kompleksne implementacije. Velika večina jih podpora samo določeno arhitekturo, kar se lahko pokaže kot pomanjkljivost. Veliko podjetij ima v svojem portfelju več naprav z različnimi arhitekturami in se razdrobljenost v razvoju kaže v povečanih stroških.

Med uporabniki operacijskega sistema Linux za vgrajene sisteme je v zadnjem obdobju postal prva izbira med zaganjalniki program z uradnim imenom Das U-Boot (the Universal Boot Loader) [21]. Značilnost zaganjalnika U-Boot je, da podpira različne procesorske arhitekture, predvsem pa ima veliko bazo aktivnih razvijalcev ter podporo proizvajalcev procesorjev in razvojnih ploščic, ki ga uporabljajo pri svojih projektih in zato aktivno sodelujejo pri nadaljnjem razvoju.

6.3.4.1 Kje in kako pridobiti U-Boot?

Razvoj U-Boota poteka kot večina odprto kodnih projektov s pomočjo `git` distribuiranega sistema za nadzor različic. Zadnja stabilna verzija programa je na voljo na spletnem naslovu http://git.denx.de/u-boot.git/.

Kot smo že omenili, je v razvoj tega programa vključenih veliko razvijalcev, predvsem pa podjetij, kar zagotavlja široko paleto podprtih razvojnih ploščic. Ker pa delamo na lastni

razvojni ploščici je potrebno U-Boot ustrezno prilagoditi in s tem zagotoviti podporo napravam iz družine LUX9.

6.3.4.2 Konfiguracija U-Boot

Ob množici podprtih procesorjev in arhitektur, kot jih zagotavlja U-Boot, je potrebno definirati tudi nekakšen sistem spreminjanja in izbire nastavitev za želeno platformo. Tako kot pri jedru Linux, se konfiguracija za U-Boot pripravi v času prevajanja, katerega rezultat je binarna slika prilagojena za specifično ciljno platformo.

Konfiguracija izbrane platforme je definirana v zaglavni datoteki (angl. header file) in določenih datotečnih povezavah, ki uparijo platformo z ustreznimi izvornimi datotekami odvisnimi od arhitekture procesorja in se avtomatsko generirajo ob izvedbi ukaza:

```
$ make <platform>_config
```

Predpona `platform` predstavlja eno izmed 800+ različnih razvojnih ploščic in konfiguracije teh ploščic, ki jih trenutno podpira U-Boot. Podprte platforme in njihove različice so navedene v datoteki `boards.cfg`. Za primer pripravimo konfiguracijo za platformo Atmel AT91SAM9260-EK :

```
$ make at91sam9260ek_dataflash_cs1_config
```

Sistem pripravi konfiguracijsko zaglavno datoteko `./include/config.h` in povezave na arhitekturo ARM, jedro ARM926EJ-S ter AT91SAM9260EK kot ciljno platformo. Naslednji korak je prilagajanje datoteke z nastavitvami za specifično platformo, ki se nahaja v `./include/configs/at91sam9260ek.h`. Najboljši vir informacij glede opisa in možnosti nastavitev v tej datoteki so komentarji razvijalcev in pa datoteka `README`, v kateri so točno opisane vse možnosti ter tudi podani primeri. V večini primerov so nastavitve, ki so podane dovolj dobro definirane za normalno delovanje razvojne plošče.

Skoraj vsi vidiki konfiguracije zaganjalnika U-Boot se lahko nastavijo skozi te mehanizme (dodajanje `CONFIG_XXX` definicij v nastavitveno datoteko). Med drugim lahko na takšen način definirano tudi dodatne funkcionalnosti, ki se bodo dodale med prevajanjem (podpora za DHCP, pomnilniški testi, podpora za razhroščevanje, podpora različnim datotečnim sistemom) in koliko ter kakšen pomnilnik je na razpolago na ploščici.

6.3.4.3 Dodajanje nove platforme

Eden od razlogov za takšno razširjenost in popularnost zaganjalnika U-Boot je v preprostosti dodajanja podpore za nove platforme. V posebno podmapo, kjer so definirane vse plošče `./board`

je potrebno dodati nastavitvene in izvorne datoteke s pripadajočo datoteko `Makefile`, ki pove kako jih je potrebno prevesti in vključiti v okolje U-Boot. Potrebno je definirati še datoteko z nastavitvami v mapi `./include/configs/`, ter definirati novo platformo v datoteki `boards.cfg`. Seveda predpostavimo, da sta naš procesor in arhitektura že podprti, za to pa obstaja kar velika verjetnost, saj U-Boot trenutno podpira večino arhitektur in različic procesorjev primernih za vgrajene sisteme. Tako si največkrat pri dodajanju nove platforme lahko pomagamo z že obstoječo, ki je naši ciljni najbolj podobna.

6.3.4.4 Dodajanje LUX-SF9 in LUX-SF9G platforme

Platforma LUX9 je strojno zelo podobna razvojnim ploščam proizvajalca Atmel AT91SAM9260-EK in Atmel AT91SAM9G20-EK. Ker je procesor SAM9G20 samo hitrejša in malce izpopolnjena zamenjava starejšega SAM9260, ju bomo podprli v okviru ene konfiguracije datoteke.

Ko smo prenesli izvorno kodo zaganjalnika U-Boot, moramo najprej pripraviti konfiguracijsko datoteko `boards.cfg`, kjer so opisane vse podprte platforme in arhitekture. Z njeno pomočjo se pripravi okolje s simboličnimi povezavami in prevedejo izvorne datoteke. Vsebina datoteke je organizirana tako, da je na prvem mestu definirano ime konfiguracije za ploščo, potem arhitektura, tip jedra, ime plošče, razvijalec plošče, tip procesorja in na koncu še nastavitve (slika 17).

Dodali bomo vrstice z našimi nastavitvami. Sledi še vpis v datoteko skrbnikov za posamezno platformo `MAINTAINERS`.

```
# Target                    ARCH       CPU         Board name          Vendor        SoC         Options
#################################################################################################################

integratorcp_cm1136         arm    arm1136    integrator     armltd     -     integratorcp
qong                        arm    arm1136    -              davedenx   mx31
at91sam9260ek_nandflash     arm    arm926ejs  at91sam9260ek  atmel      at91  at91sam9260ek:AT91SAM9260,SYS_USE_NANDFLASH
at91sam9260ek_dataflash_cs0 arm    arm926ejs  at91sam9260ek  atmel      at91  at91sam9260ek:AT91SAM9260,SYS_USE_DATAFLASH_CS0
at91sam9260ek_dataflash_cs1 arm    arm926ejs  at91sam9260ek  atmel      at91  at91sam9260ek:AT91SAM9260,SYS_USE_DATAFLASH_CS1
lux_sf9_nandflash           arm    arm926ejs  lux-sf9        evo-teh    at91  lux-sf9:AT91SAM9260,SYS_USE_NANDFLASH
lux_sf9_dataflash_cs1       arm    arm926ejs  lux-sf9        evo-teh    at91  lux-sf9:AT91SAM9260,SYS_USE_DATAFLASH_CS1
lux_sf9g_nandflash          arm    arm926ejs  lux-sf9        evo-teh    at91  lux-sf9:AT91SAM9G20,SYS_USE_NANDFLASH
lux_sf9g_dataflash_cs1      arm    arm926ejs  lux-sf9        evo-teh    at91  lux-sf9:AT91SAM9G20,SYS_USE_DATAFLASH_CS1
```

Slika 17: Dodajanje konfiguracija za platformo LUX v datoteko borads.cfg.

Ko datoteka vsebuje našo platformo, je potrebno pripraviti še podmape z nastavitvami in inicializacijo plošče, ter osnovno konfiguracijsko datoteko. Poimenovanje map in konfiguracijske datoteke mora biti v skladu z nastavitvami, ki smo jih dodali v datoteko `boards.cfg` (parametri lux). Kot predlogo za datoteko z nastavitvami lahko ponovno vzamemo že obstoječo za plošče Atmel in jo prilagodimo našim potrebam:

```
$ cp include/configs/at91sam9260ek.h include/configs/lux-sf9.h
```

- prilagodimo podporo za dva različna procesorja:

```
/* Define actual evaluation board type from used processor type */
#ifdef CONFIG AT91SAM9G20
# define CONFIG LUX SF9G    /* It's an Evo-Teh LUX-SF9G board */
#else
# define CONFIG_LUX_SF9 /* It's an Evo-Teh LUX-SF9 board */
#endif
```

- izberemo ukaze, ki jih bomo potrebovali:

```
/*
 * Command line configuration.
 */
#include <config cmd default.h>
#define CONFIG_CMD_BDI
#undef CONFIG_CMD_FPGA
#undef CONFIG CMD IMI
#undef CONFIG_CMD_IMLS
#undef CONFIG_CMD_LOADS
#undef CONFIG CMD SOURCE

#define CONFIG_CMD_MII      1
#define CONFIG CMD PING     1
#define CONFIG_CMD_DHCP     1
#define CONFIG_CMD_DNS      1
#define CONFIG CMD NAND     1
#define CONFIG_CMD_USB      1
```

- nastavimo ukaze za kopiranje jedra iz pomnilnika NAND in parametre za zagon jedra:

```
/* bootstrap + u-boot + env + linux in dataflash on CS1 */
#define CONFIG_ENV_IS_IN_DATAFLASH  1
#define CONFIG_SYS_MONITOR_BASE (CONFIG_SYS_DATAFLASH_LOGIC_ADDR_CS1 + 0x8400)
#define CONFIG_ENV_OFFSET       0x4200
#define CONFIG ENV ADDR     (CONFIG SYS DATAFLASH LOGIC ADDR CS1 + CONFIG ENV OFFSET)
#define CONFIG_ENV_SIZE     0x4200
#define CONFIG_BOOTCOMMAND  "nand read 0x22000000 0x0 0x220000; bootm"
#define CONFIG_BOOTARGS     "mem=64M console=ttyS0,115200 "         \
                            "ubi.mtd=1 root=ubi0:lux-sf9-rootfs "   \
                            "rootfstype=ubifs rw"
```

Nastavitve, ki jih lahko uporabimo v tej konfiguracijski datoteki so podrobno opisane in naštete v datoteki `README` in podmapi z dokumenti `./doc`, ki jo je predhodno priporočljivo dobro preučiti. Sedaj nam preostane samo še priprava podmape, kjer so definirane specifične inicializacijske funkcije za vsako ploščo posebej. Spet si bomo pomagali s predlogo, ki jo bomo prilagodili za module LUX9:

```
$ mkdir -p ./board/evo-teh/lux-sf9
$ cp ./board/atmel/at91sam9260-ek/* ./board/evo-teh/lux-sf9
$ mv ./boards/evo-teh/lux-sf9/at91sam9260.c ./board/evo-teh/lux-sf9/lux-sf9.c
```

Najpomembnejša funkcija v tej datoteki `lux-sf9.c` je `board_init()`(slika 18), ki skrbi za inicializacijo potrebne periferije in nastavi identifikacijo naprave (`MACH_TYPE_*`), ki jo bomo posredovali jedru Linux ob zagonu. Ta identifikacija se mora ujemati s tisto definirano v jedru, da se ta sploh zažene, drugače pade v neskončno zanko. To je varnostni mehanizem, ki preprečuje zagon napačnega jedra na napravi.

```
int board_init(void)
{
#ifdef CONFIG_LUX_SF9G
    /* arch number of LUX-SF9-Board */
    gd->bd->bi_arch_number = MACH_TYPE_LUX_SF9G;
#else
    /* arch number of LUX-SF9-Board */
    gd->bd->bi_arch_number = MACH_TYPE_LUX_SF9;
#endif
    /* adress of boot parameters */
    gd->bd->bi_boot_params = CONFIG_SYS_SDRAM_BASE + 0x100;

    at91_seriald_hw_init();
#ifdef CONFIG_CMD_NAND
    lux_sf9_nand_hw_init();
#endif
#ifdef CONFIG_HAS_DATAFLASH
    at91_spi0_hw_init((1 << 0) | (1 << 1));
#endif
#ifdef CONFIG_MACB
    lux_sf9_macb_hw_init();
#endif

    return 0;
}
```

Slika 18: Funkcija board_init().

Če smo si pri razvoju pomagali z orodjem *git*, potem lahko hitro pogledamo kakšen poseg je bil potreben (slika 19), da smo dodali podporo za novo razvojno ploščico, ki je podobna že kakšni definirani. V našem primeru je bil potreben še poseg v samo kodo U-Boot, ker smo morali odpraviti napako pri nalaganju osnovnega okolja z nastavitvenimi spremenljivkami, ki se nahajajo v pomnilniku DataFlash.

```
$ git diff master…lux-sf9
MAINTAINERS                          |    4 +
 board/evo-teh/lux-sf9/Makefile      |   60 ++++++++
 board/evo-teh/lux-sf9/led.c         |   37 +++++
 board/evo-teh/lux-sf9/lux-sf9.c     |  208 ++++++++++++++++++++++++
 board/evo-teh/lux-sf9/partition.c   |   41 ++++++
 boards.cfg                          |    6 +
 common/env_dataflash.c              |   83 ++++++++++++-----
 include/configs/lux-sf9.h           |  288 ++++++++++++++++++++++++++++++++++++
 8 files changed, 689 insertions(+), 38 deletions(-)
```

Slika 19: Definiranje nove platforme v zaganjalniku U-Boot.

Sedaj moramo pripravljeno konfiguracijo samo še izbrati in prevesti. Rezultat prevajanja izvorne kode bo datoteka `u-boot.bin`, ki se nahaja v korenu mape, pod pogojem, da smo pravilno dodali podporo za novo ploščico in se je prevajanje v binarno obliko uspešno zaključilo. Izbiro konfiguracije in prevajanje bomo izvedli z uporabo ukazov:

```
$ make lux_sf9g_dataflash_cs1_config
$ make
```

6.4 Jedro Linux

Za uspešen prenos in prilagajanje jedra Linux na želeno vgrajeno platformo je priporočeno dobro poznavanje delovanja in notranjega ustroja jedra, o katerem je na voljo precej dokumentacije [1, 5, 8]. Zelo malo pa je na voljo dokumentov, ki bi se ukvarjali s strukturo in potrebnimi informacijami, kako dodati podporo za lasten vgrajen sistem in pa katere datoteke so pomembne za arhitekturo za katero prilagajamo jedro.

V nadaljevanju se bomo posvetili organizaciji jedra in pa strukturi izvorne kode, da bomo lažje identificirali pomembno kodo, ki se ukvarja z za nas potrebno procesorsko arhitekturo ARM9 in podporo posamezni razvojni plošči. Naš končni cilj je v jedro Linux dodati podporo za platformo LUX9 in pripraviti razvojno okolje tako, da bo integracija novih modulov kar se da enostavna.

6.4.1 Verzije jedra

Uradna stran jedra Linux je http://www.kernel.org, kjer se nahajajo tudi vse glavne verzije izvorne kode jedra. Trenutna najnovejša stabilna verzija jedra je iz družine 4.x. Z verzijo jeder Linux 3.x se je Linus Torvalds odločil, da ukine manjšo revizijo in sedaj druga številka pomeni zaporedno številko jedra, tretja pa je postala verzija popravka. Pri verzijah jedra 2.6.x in starejših prva številka pomeni večjo revizijo, druga manjšo, tretja zaporedno številko znotraj družine, zadnja četrta pa verzijo popravka. Pred verzijo 2.6 jedra je druga številka pomenila produkcijsko verzijo, če je bila soda in razvojno če je bila liha [26]. Na primer:

- Linux 4.x – Produkcijsko jedro
- Linux 3.x – Produkcijsko jedro
- Linux 2.6.x – Produkcijsko jedro
- Linux 2.5.x – Razvojno jedro
- Linux 2.4.x – Produkcijsko jedro

S prihodom Linux 2.6.x jedra ne obstaja vzporedna razvojna veja, vse nove funkcionalnosti, izboljšave in popravki gredo čez serijo nadzornikov, ki filtrirajo in posredujejo spremembe glavnim upraviteljem jedra.

Za našo razvojno ploščico bomo uporabili in prilagodili jedro iz družine 2.6.x, ker zanjo obstajajo določeni neuradni popravki, gonilniki in izboljšave, ki podpirajo družino procesorjev Atmel AT91SAM9. Poleg tega je ta verzija primernejša za procesorsko šibkejše naprave.

Verzija jedra je definirana v `Makefile` datoteki, ki se nahaja v korenu mape z izvorno kodo jedra in pri naši verziji jedra 2.6.38 zgleda takole:

```
VERSION = 2
PATCHLEVEL = 6
SUBLEVEL = 38
EXTRAVERSION = .7
NAME = Flesh-Eating Bats with Fangs
```

V isti datoteki sta definirana še dva makroja, ki se uporabljata za določanje verzije jedra v kodi:

```
# Read KERNELRELEASE from include/config/kernel.release (if it exists)
KERNELRELEASE = $(shell cat include/config/kernel.release 2> /dev/null)
KERNELVERSION = $(VERSION).$(PATCHLEVEL).$(SUBLEVEL)$(EXTRAVERSION)
```

`KERNELRELEASE` predstavlja bolj popolno informacijo o verziji jedra, ki tudi vključuje oznako revizije sistema za verzioniranje kode git, ki ga uporabljajo pri razvoju jedra Linux. Ta makro je tudi vključen v samo sliko jedra. Verzijo tako lahko preverimo tudi iz konzole:

```
$ cat /proc/version
Linux version 2.6.38.7 (davidp@davidp-laptop) (gcc version 4.5.2 (Sourcery G++ Lite 2011.03-
41) ) #18 PREEMPT Thu Sep 15 22:29:01 CEST 2011
```

Za sledenje lastnih razvojnih verzij jedra lahko uporabimo polje `EXTRAVERSION` kjer dodamo poljubno oznako, recimo `-sf9`.

6.4.2 Izvorna koda

Zadnja verzija glavne veje izvorne kode jedra Linux na voljo v obliki stisnjene datoteke ali pa jo lahko naložimo s pomočjo distribuiranega sistema za nadzor različic. Za družino 2.6.x jedra Linux naredimo to takole:

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git linux-2.6
```

Ker podpora za vse naprave in funkcije, ki jih omogoča procesor AT91SAM9, še niso vključene v jedro, si bomo pomagali s popravki in dodatki, ki so na voljo na različnih forumih, kjer

sodelujejo razvijalci. Za osnovo bomo vzeli jedro Linux 2.6.38.7, ter dodali popravke za arhitekturo AT91 s strani [22] in pa foruma [23], namenjenega izmenjavi informacij za omenjeno arhitekturo. Spisali bomo še svoje popravke in s tem v jedro dodali podporo za našo družino LUX9 razvojnih ploščic.

Zavedati se moramo, da čeprav jedro Linux podpira zelo veliko število arhitektur in verzij čipov za vgrajene sisteme, zaradi velikosti projekta in pa strogih meril glede kvalitete kode vsi gonilniki in izboljšave ne pridejo takoj v glavno vejo jedra. Vsak resnejši proizvajalec čipov zato vzdržuje lasten repozitorij ali pa vsaj skupek popravkov, ki omogočajo celovitejšo podporo lastnim čipom, glede na originalno jedro in ga je priporočljivo poiskati na proizvajalčevi strani ali razvojnih forumih. Šele tako pripravljeno jedro je smotrno uporabiti za nadaljnji razvoj in prilagajanje lastnemu produktu.

Vrhnja korenska mapa izvorne kode jedra, ki smo jo prenesli s pomočjo orodja `git` ali pa razširili arhivsko datoteko vsebuje naslednje podmape (zaradi preglednosti smo izpustili vse datoteke in mape, ki so povezane z okoljem za prevajanje jedra):

```
arch/           drivers/   init/    mm/       security/  virt/
block/          firmware/  ipc/     net/      sound/
crypto/         fs/        kernel/  samples/  tools/
Documentation/  include/   lib/     scripts/  usr/
```

Večina teh map vsebuje še več nivojev podmap z izvorno kodo, datotekami `Makefile` in konfiguracijskimi datotekami. Od vseh je najbolj obsežna mapa `./drivers`, kjer se nahajajo gonilniki za vse vrste naprav od mrežnih kartic, krmilnikov USB, blokovnih naprav itd. Naslednja mapa po velikosti je `./arch` kjer se nahaja podpora za 20 različnih procesorskih arhitektur in še veliko več izpeljank družin procesorjev ter razvojnih ploščic. V tej mapi bomo našli tudi podporo za naš procesor ter dodali podporo za našo družino modulov LUX9.

Od datotek, ki smo jih zgoraj izpustili, moramo posebej omeniti skrito nastavitveno datoteko, ki določa katere komponente bodo vključene v jedro in pa še dve datoteki, ki nastaneta ob uspešnem postopku prevajanja jedra in sicer `System.map` ter `vmlinux`. Obe sta za nas zelo pomembni in si ju bomo bolj podrobno ogledali.

6.4.3 Prevajanje jedra Linux

Proces prevajanja jedra Linux kreira nekaj datotek, ki so neodvisne od izbrane arhitekture in so za vse enake, to velja za zgoraj omenjeni `System.map` in `vmlinux`, ter eno ali več arhitekturno odvisnih binarnih modulov. Datoteka `System.map` vsebuje listo povezanih objektov in njihovih

naslovov, ki jih je povezovalnik povezal v arhitekturno odvisno binarno ELF[2] datoteko `vmlinux`. V praksi se ta datoteka skoraj nikoli ne uporabi za zagon računalnika ali naprave, čeprav bi bilo to mogoče. Za zagon se uporablja rahlo prilagojena in stisnjena verzija binarne oblike jedra [1].

Kako pridemo do teh dveh datotek? Ko smo pripravili jedro Linux z vso potrebno podporo za našo ploščico smo v postopku ustvarili tudi privzeto konfiguracijsko datoteko, ki se bo uporabila pri prevajanju jedra. Ker prevajamo za arhitekturo, ki je različna od tiste na kateri razvijamo, v našem primeru prevajamo na arhitekturi x86 za arhitekturo ARM, moramo imeti nameščeno ustrezno navzkrižno razvojno okolje. Takšno okolje si lahko pripravimo in prevedemo sami, lahko pa uporabimo katerega od že pripravljenih. Ker pa je to precej zahtevno početje, se bomo poslužili okolja, ki ga pripravlja podjetje Mentor Graphics in je na voljo tudi na naslovu http://www.luxboards.com/pub/LUX9/software/tools. Namestitev zahteva samo razširitev stisnjene arhivske datoteke.

Jedro Linux za platformo LUX9 prevedemo tako, da orodju make navedemo konfiguracijsko datoteko za katero platformo želimo prevesti jedro, potem pa še v kakšni izhodni obliki želimo prevedeno jedro zapakirati. Ustrezni ukazi so prikazani na sliki 20.

```
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- lux_sf9_defconfig
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- uImage
```

Slika 20: Konfiguracija in prevajanje jedra Linux.

Parametri `ARCH=arm` povedo sistemu za prevajanje za katero arhitekturo prevajamo jedro, `CROSS_COMPILE` pa kateri navzkrižni prevajalnik in orodja naj pri tem uporabi. V prvi vrstici smo izbrali še pripravljene privzete nastavitve za ciljno platformo, v drugi pa povedali v kakšni obliki želimo imeti končni produkt prevajanja. Ne glede na izbrano obliko se generirata datoteki `System.map` in `vmlinux`, iz katere izhajajo vse ostale različice binarne slike jedra primernega za nalaganje na modul. Rezultati uspešnega prevajanja in izhodne datoteke so prikazane na sliki 21.

[2] Executable and Linking Format (standardni format za izvedbene binarne datoteke v Linux okolju)

```
  LD      vmlinux.o
  MODPOST vmlinux.o
  GEN     .version
  CHK     include/generated/compile.h
  UPD     include/generated/compile.h
  CC      init/version.o
  LD      init/built-in.o
  LD      .tmp_vmlinux1
  KSYM    .tmp_kallsyms1.S
  AS      .tmp_kallsyms1.o
  LD      .tmp_vmlinux2
  KSYM    .tmp_kallsyms2.S
  AS      .tmp_kallsyms2.o
  LD      vmlinux
  SYSMAP  System.map
  SYSMAP  .tmp_System.map
  OBJCOPY arch/arm/boot/Image
  Kernel: arch/arm/boot/Image is ready
  AS      arch/arm/boot/compressed/head.o
  LZO     arch/arm/boot/compressed/piggy.lzo
  AS      arch/arm/boot/compressed/piggy.lzo.o
  CC      arch/arm/boot/compressed/misc.o
  CC      arch/arm/boot/compressed/decompress.o
  SHIPPED arch/arm/boot/compressed/lib1funcs.S
  AS      arch/arm/boot/compressed/lib1funcs.o
  LD      arch/arm/boot/compressed/vmlinux
  OBJCOPY arch/arm/boot/zImage
  Kernel: arch/arm/boot/zImage is ready
  UIMAGE  arch/arm/boot/uImage
Image Name:   Linux-2.6.38.7
Created:      Sun Oct  2 14:56:44 2011
Image Type:   ARM Linux Kernel Image (uncompressed)
Data Size:    1965344 Bytes = 1919.28 kB = 1.87 MB
Load Address: 20008000
Entry Point:  20008000
  Image arch/arm/boot/uImage is ready
```

Slika 21: Rezultat prevajanja jedra Linux.

6.4.4 Datoteka: vmlinux

Datoteka `vmlinux` je samostojna slika monolitnega jedra v formatu ELF, kar pomeni, da so razrešeni vsi sklici na zunanje povezave oziroma objekte. Če to sliko zaženemo v pravilnem okolju in kontekstu z zaganjalnikom, ki je namenjen zagonu jedra Linux, bo rezultat delujoča naprava pod nadzorom jedra Linux.

Da bomo lažje razumeli delovanje sistema kot celote in spoznali njegove sestavne dele se moramo osredotočiti na zgradbo jedra `vmlinux` kot binarnega objekta v ELF formatu. To bomo naredili najlažje tako, da si ogledamo, kaj se skriva za ukazom `LD vmlinux` pri prevajanju jedra, kar pomeni povezovanje posameznih objektov v binarno sliko jedra prikazano na sliki 22:

```
arm-none-linux-gnueabi-ld -EL -p --no-undefined -X --build-id -o vmlinux \
-T arch/arm/kernel/vmlinux.lds  \
arch/arm/kernel/head.o          \
arch/arm/kernel/init_task.o     \
init/built-in.o                 \
--start-group                   \
  usr/built-in.o                \
  arch/arm/kernel/built-in.o    \
  arch/arm/mm/built-in.o        \
  arch/arm/common/built-in.o    \
  arch/arm/mach-at91/built-in.o \
  kernel/built-in.o             \
  mm/built-in.o                 \
  fs/built-in.o                 \
  ipc/built-in.o                \
  security/built-in.o           \
  crypto/built-in.o             \
  block/built-in.o              \
  arch/arm/lib/lib.a            \
  lib/lib.a                     \
  arch/arm/lib/built-in.o       \
  lib/built-in.o                \
  drivers/built-in.o            \
  sound/built-in.o              \
  firmware/built-in.o           \
  net/built-in.o                \
--end-group .tmp_kallsyms2.o
```

Slika 22: Povezovanje objektov v binarno obliko jedra.

Kot lahko opazimo iz klica povezovalnika, je slika `vmlinux` sestavljena iz večih binarnih slik objektov, ki so povezane v celoto, in ji pravimo jedro Linux. Parametra povezovalnika sta izhodna datoteka `-o vmlinux` in pa povezovalna skripta `-T vmlinux.lds`, ki vsebuje detajlna navodila kako več binarnih slik objektov sestaviti v delujoče jedro. Da bi lažje razumeli vse komponente in s tem tudi strukturo izvorne kode jih navedli in podrobneje opisali v tabeli 8 [1].

Tabela 8: Objekti za povezovanje.

Objekt (komponenta)	Opis
arch/arm/kernel/head.o	Arhitekturno specifična zagonska koda. Objekt je bil preveden iz ../arch/arm/kernel/head.S datoteke napisane v zbirniku, kjer se nahaja prva vrstica kode, ki jo zažene jedro.
arch/arm/kernel/init_task.o	Začetna nit in strukture potrebne za inicializacijo opravil jedra.
init/built-in.o	Glavna inicializacijska koda jedra.
usr/built-in.o	Vgrajena slika initramfs, če je definirana.
arch/arm/kernel/built-in.o	Arhitekturno specifična koda jedra.
arch/arm/mm/built-in.o	Arhitekturno specifična koda za upravljanje s pomnilnikom.
arch/arm/common/built-in.o	Arhitekturno specifična generična koda.
arch/arm/mach-at91/built-in.o	Strojno specifična koda za procesor in ploščico, po navadi inicializacija periferne strojne opreme. V tem delu smo dodali podporo za LUX9 platformo.
kernel/built-in.o	Skupne komponente jedra.
mm/built-in.o	Skupne komponente za upravljanje s pomnilnikom.
fs/built-in.o	Koda za podporo datotečnim sistemom.
ipc/built-in.o	Medprocesna komunikacija.
security/built-in.o	Varnostne komponente jedra.
crypto/built-in.o	Kriptografski sistemski klici.
block/built-in.o	Podpora za blokovne naprave (diski, MTD)
arch/arm/lib/lib.a	Arhitekturno specifični skupni objekti.
lib/lib.a	Skupne pomožne funkcije jedra.
arch/arm/lib/built-in.o	Arhitekturno specifične pomožne funkcije jedra.
lib/built-in.o	Knjižnica skupnih funkcij.
drivers/built-in.o	Vsi vključeni gonilniki brez modulov.
sound/built-in.o	Gonilniki za zvok.
firmware/built-in.o	Objekti gonilnikov vdelane programske opreme (firmware).
net/built-in.o	Podpora mrežnim protokolom.
.tmp_kallsyms2.o	Tabela simbolov jedra.

6.4.5 Datoteka slike sestavljenega jedra Linux

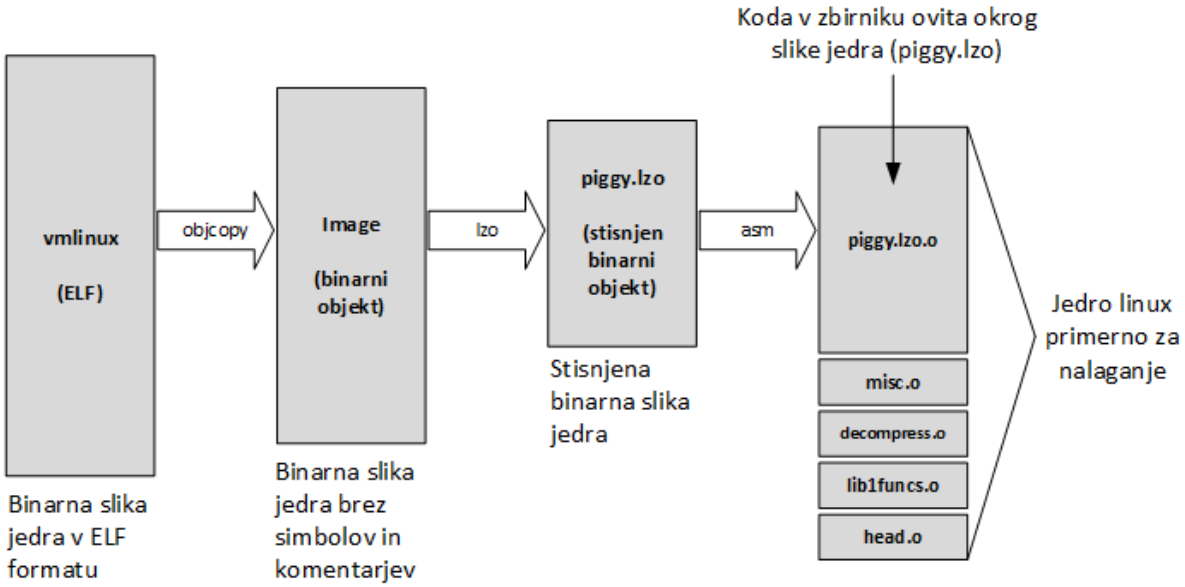
V prejšnjem poglavju smo opisali kako je zgrajena datoteka monolitnega jedra Linux `vmlinux` v ELF formatu, ki jo lahko direktno naložimo in poženemo na določenih arhitekturah po konverziji v čisto binarno obliko, primer takšne je arhitektura PowerPC. Naša izbrana arhitektura ARM tega ne omogoča in je potrebno to obliko slike jedra ustrezno prilagoditi in dopolniti, da jo bo lahko zaganjalnik naložil in zagnal.

Po fazi povezovanja modulov v ELF datoteko `vmlinux`, sledi še procesiranje in prevajanje nekaj dodatnih modulov, kot lahko vidimo v tabeli 9. Med temi so datoteke `head.o`, `piggy.lzo.o` in so specifične za vsako arhitekturo (v našem primeru ARM) in vsebujejo nizkonivojska orodja in funkcije, ki so potrebne za zagon jedra za posamezno arhitekturo [10].

Tabela 9: Objekti jedra.

Objekt (komponenta)	Opis
`Vmlinux`	Slika jedra v ELF formatu, vsebuje vse simbole, komentarje, informacije za razhroščevanje (prevedeno s stikalom –g) in arhitekturno specifične komponente.
`System.map`	Tekstovna datoteka s tabelo simbolov jedra.
`Image`	Binarna slika jedra brez simbolov, komentarjev, kode za razhroščevanje.
`head.o`	Generična zagonska koda specifična za ARM arhitekturo. Zaganjalnik preda kontrolo temu objektu.
`piggy.lzo`	Datoteka `Image` stisnjena z algoritmom `lzo`. (v postopku konfiguracije se lahko izbere več različnih tipov stiskanja)
`piggy.lzo.o`	Datoteka `piggy.lzo` v obliki, ki se lahko povezuje z naslednjimi objekti (`misc.o`)
`misc.o`	Dekompresija slike jedra (`piggy.lzo`), preverjanje šifre modela naprave in izpis »`Uncompressing Linux ...`« preko serijskih vrat.
`decompress.o`	Specifične funkcije za dekompresijo glede na izbran algoritem.
`lib1funcs.o`	Optimizirane funkcije deljenja spisane v zbirnem jeziku.
`Vmlinux`	Datoteka sestavljenega jedra, ime je isto kot za sliko jedra v formatu ELF, vendar ne gre za isto datoteko.
`zImage`	Končna slika jedra, ki jo naloži in zažene zaganjalnik.

Na sliki 23 je grafično predstavljen postopek izgradnje datoteke `zImage`, ki jo potem naloži in zažene zaganjalnik.



Slika 23: Izgradnja sestavljenega jedra Linux.

6.4.5.1 Objektna datoteka Image

Potem, ko se pripravi datoteka `vmlinux` v ELF obliki, sistem za prevajanje in sestavljanje jedra nadaljuje s procesiranjem objektov in datotek navedenih v tabeli 9. Objektno datoteko `Image` kreira iz datoteke `vmlinux` , tako da odstrani podvojene sekcije (opombe in komentarje, možnost –R .comment) ter simbole za razhročevanje (opcija –S), če so bili prisotni (slika 24).

```
arm-none-linux-gnueabi-objcopy -O binary -R .comment -S  vmlinux arch/arm/boot/Image
```

Slika 24: Kopiranje jedra v ELF obliki v čisto binarno obliko.

6.4.5.2 Arhitekturno specifične objektne datoteke

Če naprej sledimo procesu sestavljanja jedra, primernega za nalaganje na ciljno platformo, potem sledi stiskanje objektne datoteke `Image` z izbranim kompresijskim algoritmom, v našem primeru je to lzo, na izbiro sta še gzip in lzma, vsak s svojimi prednostmi in slabostmi. Mi smo se odločili za lzo, ker omogoča najhitrejšo dekompresijo. Stiskanje datoteke `Image` proizvede stisnjeno objektno datoteko `piggy.lzo` (25):

```
cat arch/arm/boot/compressed/../Image | lzop -9 && printf \144\341\066\000) >
arch/arm/boot/compressed/piggy.lzo
```

Slika 25: Stiskanje binarne oblike jedra z algoritmom lzo.

Sledi kreiranje objektne datoteke `piggy.lzo.o`, primerne za kasnejše povezovanje z ostalimi objektnimi datotekami v tabeli 9. Prevede se datoteka `piggy.lzo.S`, napisana v zbirniku, ki vsebuje referenco na `piggy.lzo` datoteko, ki je vključena (angl. piggybacked) kot tovor v zbirnem jeziku spisan nalagalnik bootstrap. Naloga nalagalnika je, da poskrbi za pravilno inicializacijo procesorja in pomnilniških struktur, razširi binarno sliko jedra, ga naloži na pravilen naslovni prostor v sistemskem pomnilniku in nazadnje preda jedru Linux nadzor. Vsebina `piggy.lzo.S` datoteke je prikazana na sliki 26 in je zelo preprosta.

```
        .section .piggydata,#alloc
        .globl  input_data
input_data:
        .incbin "arch/arm/boot/compressed/piggy.lzo"
        .globl  input_data_end
input_data_end:
```

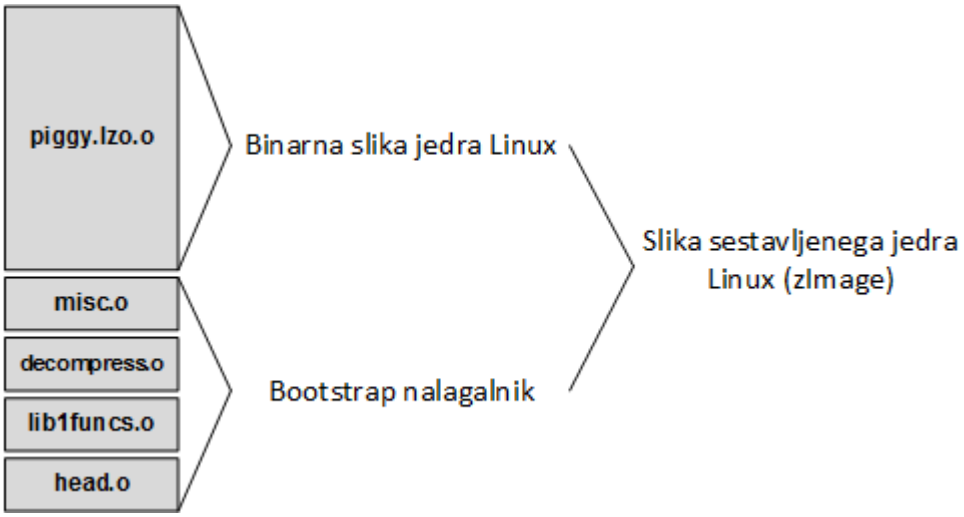
Slika 26: Vsebina datoteke piggy.lzo.S v zbirniku.

Med sekcijama `input_data` in `input_data_end`, ki jih nalagalnik bootstrap uporabi kot identifikacijo začetka in konca tovora (jedro Linux) je direktiva `.incbin`, ki je nekakšen zbirniški `#include` samo, da gre v tem primeru za binarno sliko datoteke.

6.4.5.3 Nalagalnik Bootstrap

Veliko arhitektur uporablja bootstrap nalagalnik kot drugostopenjski nalagalnik za kopiranje jedra Linux v pomnilnik. Nekateri preverijo kontrolno vsoto (angl. checksum) slike jedra, večina pa jih samo dekompresira in prestavi sliko jedra na ustrezno lokacijo v pomnilniku. Razlika med zaganjalnikom kot je U-Boot in nalagalnikom bootstrap je preprosta, prvi nadzira delovanje strojne opreme po priključitvi napajanja in za delovanje ne potrebuje jedra Linux, funkcija drugega pa je kot neke vrste lepilo med zaganjalnikom in jedrom Linux. Naloga nalagalnika bootstrap je, da v okviru svojega delovanja pripravi okolje v katerem se bo zagnalo jedro Linux in dekompresira ter prestavi sliko jedra na predvideno lokacijo v pomnilniku.

Koncept nalagalnika bootstrap je grafično lepo predstavljen na sliki 27.



Slika 27: Sestavljena slika jedra Linux.

Funkcije, ki jih nalagalnik bootstrap izvede so:

- Nizkonivojska inicializacija procesorja napisana v zbirniku, ki vključuje podporo za vklop internega podatkovnega in ukaznega predpomnilnika, izklop prekinitev in pripravo okolja za C programe (`head.o`).
- Dekompresija in kopiranje na ustrezno lokacijo v pomnilniku (`misc.o`, `decompress.o`, `lib1funcs.o`).

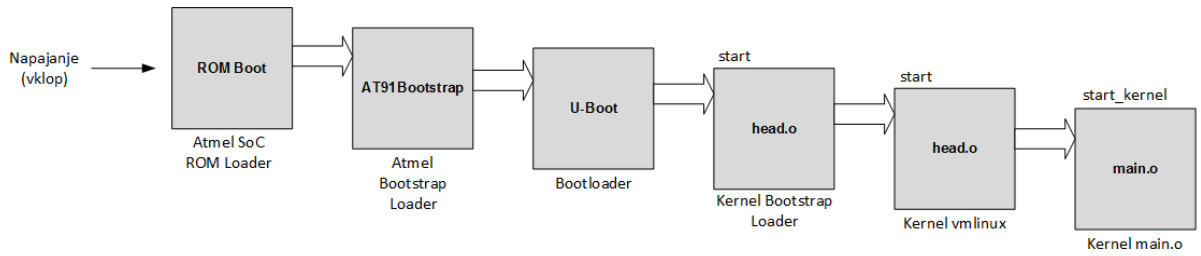
Omeniti je potrebno, da so ti detajli specifični za arhitekturo ARM. Druge arhitekture recimo x86, PowerPC nalagalnika bootstrap ne potrebujejo. S podrobno analizo gradnje jedra Linux, kot smo jo predstavili v tem poglavju, lahko sestavimo podrobno sliko procesa zagona jedra Linux za vse podprte arhitekture [13].

6.4.6 Prenos kontrole med stopnjami zagona

Sedaj, ko dobro poznamo zgradbo in strukturo sestavljene slike jedra Linux, se lahko posvetimo še celotnemu procesu zagona platforme od priklopa napajanja pa do zagonu prvega procesa `init` operacijskega sistema. Posebej smo opisali nalagalnike sistema od prvostopenjskega, ki je preprost in prevzame nadzor takoj po priklopu napajanja, do drugostopenjskega U-Boot, ki skrbi za osnovno inicializacijo strojne opreme, nalaganje jedra operacijskega sistema, ter nudi osnovno diagnostiko stojne opreme.

Razvojna platforma iz družine LUX9 temelji na procesorjih Atmel AT91SAM9, ki so zasnovani na osnovi arhitekture ARM9. Zaradi uporabe signala BMS se najprej zažene notranji BootROM program, ki na zunanjih pomnilnikih poišče prvostopenjski nalagalnik v našem primeru AT91Bootstrap. Ta mora biti dovolj preprost in majhen, da lahko v celoti teče v procesorjevem internem pomnilniku SRAM. Njegova naloga je, da inicializira zunanji pomnilnik SD/DDR RAM ter vanj prekopira drugostopenjski bootloader program (U-Boot) in mu preda nadzor. Celoten proces je podrobneje opisan v poglavjih 5.1.2 in 6.3.

Ta potem naloži iz različnih podprtih medijev sestavljeno sliko jedra Linux na določeno lokacijo v pomnilniku in preda nadzor modulu `head.o` na oznaki `start` nalagalnika bootstrap, ki je sestavni del slike jedra. Celoten potek je lepo prikazan na sliki 28.



Slika 28: Zagonski cikel na procesorju Atmel AT91SAM9.

Glavna naloga nalagalnika jedra bootstrap je, da pripravi ustrezno okolje, dekompresira in prestavi jedro na ustrezno mesto v pomnilniku, ter mu na koncu preda nadzor. Nadzor po svoji opravljeni funkciji direktno preda jedru, specifično modulu `head.o` jedra, ki je enak za večino arhitektur. Ob tem nas ne sme zavesti enako ime modula nalagalnika in samega jedra. Modul jedra `head.o` potem, ko opravi arhitekturno specifične naloge, preda dokončno nadzor funkciji `start_kernel()`, ki se nahaja v datoteki `main.c`.

6.4.6.1 Vstopna točka v jedro: head.o

Cilj razvijalcev jedra je bil spisati arhitekturo specifičen modul `head.o` tako, da bi bil čim bolj generičen in neodvisen od strojne platforme, v našem primeru razvojne ploščice ali naprave. Vstopna točka je torej enaka za vse procesorje znotraj arhitekture in strojne platforme. Modul izvira iz datoteke napisane v zbirnem jeziku `head.S` in se nahaja v mapi `../arch/<ARCH>/kernel/head.S`, kjer je `<ARCH>` arhitektura za katero razvijamo, v našem primeru `ARCH=arm`.

Modul `head.o` opravi inicializacijo, ki je arhitekturno in tudi procesorsko specifična, kot predpripravo za izvajanje glavnega dela jedra Linux. Inicializacija stojne platforme se izvede in definira na drugem mestu. Med drugimi `head.o` izvede naslednje naloge:

- preveri pravilnost verzije procesorja in arhitekture za katero je bil preveden;
- pripravi strukturo ostranjevalne tabele;
- vključi enoto za upravljanje z navideznim pomnilnikom (MMU);
- pripravi osnovno ogrodje za zaznavanje napak in poročanje;
- skoči na začetek glavnega dela jedra `start_kernel()` v datoteki `main.c`.

Glavna naloga tega modula je preverjanje pravilnosti arhitekture in strojne platforme, na kateri bo teklo jedro, ter priprava in inicializacija strukture, potrebne za delovanje glavnega dela jedra Linux, torej preklop iz fizičnega naslavljanja pomnilnika v navideznega s pomočjo krmilnika in preslikovalnika navideznega pomnilnika (enote MMU). Za procesorje, ki ne premorejo enote MMU, obstaja posebna različica jedra in distribucije Linux z imenom uClinux [13].

6.4.6.2 Zagon jedra: main.c

Zadnje dejanje modula jedra `head.o` je, da preda nadzor zagonski datoteki glavnega dela jedra Linux. Vsaka arhitektura ima drugačno sintakso in metodologijo kako predati nadzor glavnemu delu jedra. V primeru arhitekture ARM je to definirano v datoteki `head.S`, bolj natančno v datoteki `head-common.S`, v katero so razvijalci razdvojili skupno kodo in je dodana v glavno datoteko z ukazom `include`. Klic je izveden na način:

```
b   start_kernel
```

S tem ukazom je predan nadzor s strani modula `head.o` funkciji `start_kernel()`, napisani v jeziku C, ki se nahaja v datoteki `../init/main.c` in je ista za vse arhitekture.

Kdor želi bolj podrobno spoznati in razumeti delovanje jedra Linux, iz katerih komponent je sestavljeno in kako se te komponente povezujejo med sabo, mora začeti s to datoteko. Vsebuje namreč večino inicializacije, ki ni napisana v zbirnem jeziku, od kreiranja prve procesne niti pa do priklopa korenskega datotečnega sistema in zagona prve aplikacije v uporabniškem načinu.

Ena izmed prvih stvari, ki se zgodijo v funkciji `start_kernel()`, je klic funkcije `setup_arch(&command_line)`, ki je v našem primeru definirana v datoteki `../arch/arm/kernel/setup.c`. Edini parameter, ki ga ta funkcija sprejme je kazalec na strukturo ukazne vrstice jedra.

Za nas najbolj pomemben del te funkcije, poleg identificiranja specifičnega procesorja znotraj arhitekture, je inicializacija strojne platforme, v našem primeru modulov in ploščic družine LUX9. Ta mehanizem se razlikuje med arhitekturami. Za procesorje ARM se nahaja v mapah `./arch/arm/mach-*` odvisno od procesorja. Mi smo lastno platformo in inicializacijo periferne strojne opreme na vsaki ploščici posebej definirali v mapi `./arch/arm/mach-at91` v datotekah, ki se začnejo s predpono `board-*.c` . Vsak modul ima svojo inicializacijsko datoteko, ki je navedena v tabeli 10.

Tabela 10: Inicializacijske datoteke posameznih modulov LUX9.

Modul	Mikroprocesor	Inicializacijska datoteka
LUX-SF9	AT91SAM9260	`board-lux-sf9.c`
LUX-SF9G	AT91SAM9G20	`board-lux-sf9g.c`
LUX-SFX9	AT91SAM9263	`board-lux-sfx9.c`
LUX-EDK9	AT91SAM9260	`board-lux-edk9.c`

Prva programska nit, ki jo jedro kreira se imenuje `init()` in dobi številko procesa 1 oziroma PID 1. Kot vemo iz delovanja operacijski sistemov Unix, postane ta proces oče vseh ostalih procesov v uporabniškem okolju. Ta proces zažene prvi program v uporabniškem načinu, ki nadaljuje z inicializacijo platforme v uporabniškem načinu (slika 29).

```
/*
 * We try each of these until one succeeds.
 *
 * The Bourne shell can be used instead of init if we are
 * trying to recover a really broken machine.
*/
if (execute_command) {
        run_init_process(execute_command);
        printk(KERN_WARNING "Failed to execute %s.  Attempting "
                                    "defaults...\n", execute_command);
}
run_init_process("/sbin/init");
run_init_process("/etc/init");
run_init_process("/bin/init");
run_init_process("/bin/sh");

panic("No init found.  Try passing init= option to kernel. "
      "See Linux Documentation/init.txt for guidance.");
```

Slika 29: Zagon prvega procesa v uporabniškem načinu.

6.4.7 Sistem gradnje jedra

Sistem za gradnjo in konfiguracijo jedra je precej kompleksen, tako kot je tudi pričakovati od programske opreme, ki obsega več kot 10 milijonov vrstic kode, vendar ga moramo poznati, če želimo prilagoditi okolje za lastne potrebe in dodati novo strojno platformo.

V korenu mape, kjer se nahaja izvorna koda jedra, in še v večini ostalih podmap lahko najdemo dve datoteki: `Makefile` in `Kconfig`, ki sta ključnega pomena pri konfiguraciji in gradnji jedra, načrt in nastavitve kako zgraditi jedro po naših željah pa se zapiše v datoteko `.config`, ki se tudi nahaja v korenski mapi [1].

6.4.7.1 Datoteka .config

Datoteka `.config` predstavlja načrt kako zgraditi sliko jedra Linux po naših specifikacijah. Normalno je, da na začetku vložimo veliko energije in napora za pripravo želene konfiguracije za lastno vgrajeno platformo. Obstaja več možnosti in načinov kako pripravimo in urejamo to datoteko od grafičnih do tekstovnih urejevalnikov, ki na koncu našega urejanja generirajo omenjeno datoteko z nastavitvami v korenski mapi.

Ko smo enkrat zadovoljni z izbranimi nastavitvami, je potrebno pripraviti privzeto konfiguracijsko datoteko za specifično platformo, v našem primeru za LUX9, tako da kopiramo datoteko `.config` v podmapo `arch/arm/configs/` z imenom `lux_sf9_defconfig`. Tako nam ukaz `$make lux_sf9_defconfig` povrne nastavitve za našo platformo v naše prvotno stanje. Enako storimo, če smo morali počistiti okolje za gradnjo jedra z ukazon `$make distclean` ali `$make mrproper` zaradi testiranja.

Zgradba datoteke je dokaj preprosta in predstavlja zbirko konfiguracijskih možnosti, ki jih lahko vključimo v proces prevajanja ali ne. Zavedati se moramo, da ima jedro Linux monolitno

strukturo, kar pomeni, da na koncu procesa prevajanja in povezovanja dobimo samostojen statično povezan program. Kljub temu pa je možno prevesti in postopno povezati skupek izvorne kode v samostojen objekt, primeren za dinamično vstavljanje v jedro, ki se izvaja. Takšni objekti so v Linuxu poznani kot moduli, ki jih lahko priključimo v delujoče jedro, največkrat so to gonilniki naprav. Ko je jedro naloženo, za vstavljanje in odstranjevanje teh modulov skrbijo posebni programi.

```
…
CONFIG_MMC=y
# CONFIG_MMC_DEBUG is not set
# CONFIG_MMC_UNSAFE_RESUME is not set
# CONFIG MMC CLKGATE is not set
# MMC/SD/SDIO Card Drivers
CONFIG_MMC_BLOCK=y
CONFIG_MMC_BLOCK_MINORS=8
CONFIG_MMC_BLOCK_BOUNCE=y
# CONFIG_MMC_TEST is not set
# MMC/SD/SDIO Host Controller Drivers
# CONFIG_MMC_SDHCI is not set
CONFIG_MMC_AT91=m
# CONFIG_MMC_ATMELMCI is not set
# CONFIG_MMC_SPI is not set
# CONFIG_MMC_DW is not set
# CONFIG_MMC_USHC is not set
…
```

Slika 30: Konfiguracijska datoteka jedra.

Če si ogledamo delček konfiguracijske datoteke `.config` (slika 30) lahko opazimo, da se bo gonilnik za krmilnik kartic SD/MMC (`CONFIG_MMC_AT91=m`) v našem čipu prevedel kot modul, primeren za dinamično vstavljanje. Druga izbira bi bila `=y`, kar bi pomenilo, da bi bil gonilnik preveden in statično vključen sliko jedra samega. Končal bi v objektu `drivers/built-in.o`, ki ga povezovalnik poveže z ostalimi v monolitno jedro.

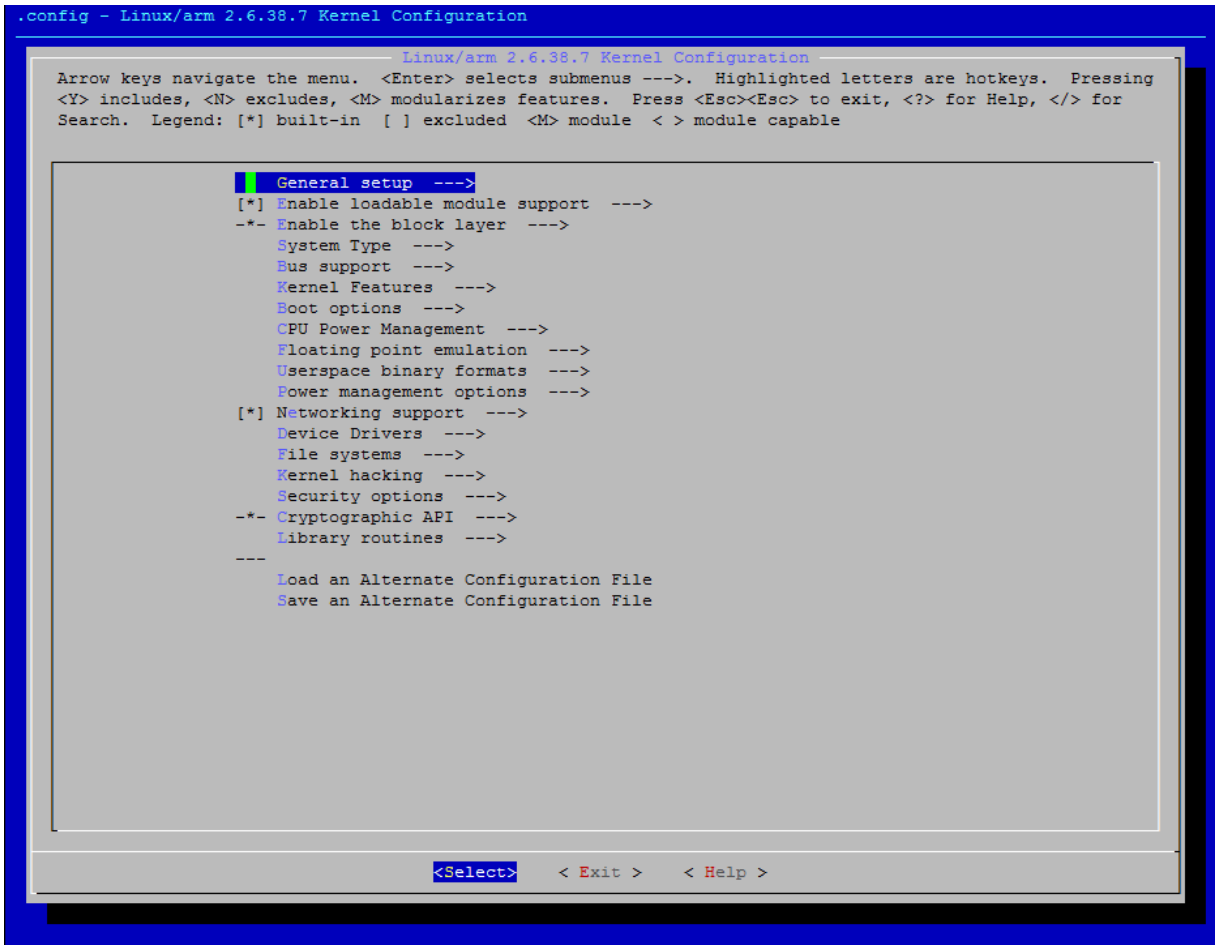
Za urejanje konfiguracijske datoteke, imamo več možnosti od čisto tekstnega načina, kjer moramo odgovoriti po vrsti na vse možnosti, ki so na izbiro, do grafičnih urejevalnikov v okenskem sistemu X. Nekaj najbolj pogostih je:

- `config` – posodobi nastavitve z vrstično orientiranim programom;
- `menuconfig` – posodobi nastavitve s pomočjo menijskega programa v konzoli;
- `xconfig` – posodobi nastavitve s pomočjo programa napisanega za QT okolje;
- `gconfig` – posodobi nastavitve s pomočjo programa napisanega za GTK+ okolje.

Program zaženemo tako, da v konzoli v korenu mape z izvorno kodo jedra Linux napišemo:

```
$make menuconfig
```

Prikaže se nam menijski program (slika 31) za konfiguriranje nastavitev jedra. Ob končani konfiguraciji se zapiše datoteka `.config`.



Slika 31: Menijski program za konfiguracijo jedra.

6.4.7.2 Datoteka Kconfig

Datoteka `Kconfig` je odgovorna za proces konfiguracije funkcij in lastnosti znotraj svojih podmap izvorne kode jedra Linux. Vsebino datotek `Kconfig` interpretira in razčleni konfiguracijski podsistem, ki jo potem predstavi uporabniku kot spisek nastavitvenih možnosti s pripadajočo pomočjo.

Konfiguracijski nastavitveni program, recimo `menuconfig`, prebere datoteke `Kconfig`, začenši s tisto v podmapi `arch` (odvisno od arhitekture za katero prevajamo jedro). Datoteka vsebuje veliko vrstic, ki zgledajo takole: `source "arch/arm/mach-at91/Kconfig"`.

Ta ukaz pove urejevalniku, da naj prebere še eno konfiguracijsko datoteko iz druge lokacije. Vsaka arhitektura ima veliko takšnih ukazov v datotekah `Kconfig`, ki skupaj tvorijo kompleksno zbirko menijev, možnosti in pomoči za uporabnika ob nastavljanju jedra za želeno arhitekturo oziroma platformo.

Sedaj ko poznamo na kakšen način deluje nastavljanje opcij jedra, bomo na podoben način dodali še podporo za platformo LUX9. Dodali bomo konfiguracijo, specifično za modul LUX-SF9. Ker družina LUX9 temelji na družini procesorjev AT91, bomo podporo dodali v konfiguracijski datoteki `arch/arm/mach-at91/Kconfig` pod določenim modelom procesorja, kot je prikazano na sliki 32.

```
if ARCH_AT91SAM9260

comment "AT91SAM9260 Variants"

config ARCH_AT91SAM9260_SAM9XE
        bool "AT91SAM9XE"
        help
          Select this if you are using Atmel's AT91SAM9XE System-on-Chip.
          They are basically AT91SAM9260s with various sizes of embedded Flash.

comment "AT91SAM9260 / AT91SAM9XE Board Type"

config MACH_AT91SAM9260EK
        bool "Atmel AT91SAM9260-EK / AT91SAM9XE Evaluation Kit"
        select HAVE_AT91_DATAFLASH_CARD
        select HAVE_NAND_ATMEL_BUSWIDTH_16
        help
          Select this if you are using Atmel's AT91SAM9260-EK or AT91SAM9XE Evaluation Kit
          <http://www.atmel.com/dyn/products/tools_card.asp?tool_id=3933>

config MACH_CAM60
        bool "KwikByte KB9260 (CAM60) board"
        help
          Select this if you are using KwikByte's KB9260 (CAM60) board based on the Atmel
AT91SAM9260.
          <http://www.kwikbyte.com/KB9260.html>

config MACH_SAM9_L9260
        bool "Olimex SAM9-L9260 board"
        select HAVE_AT91_DATAFLASH_CARD
        help
          Select this if you are using Olimex's SAM9-L9260 board based on the Atmel
AT91SAM9260.
          <http://www.olimex.com/dev/sam9-L9260.html>
config MACH_AFEB9260
        bool "Custom afeb9260 board v1"
        help
          Select this if you are using custom afeb9260 board based on
          open hardware design. Select this for revision 1 of the board.
          <svn://194.85.238.22/home/users/george/svn/arm9eb>
          <http://groups.google.com/group/arm9fpga-evolution-board>

config MACH_USB_A9260
        bool "CALAO USB-A9260"
        help
          Select this if you are using a Calao Systems USB-A9260.
          <http://www.calao-systems.com>

config MACH_QIL_A9260
        bool "CALAO QIL-A9260 board"
        help
          Select this if you are using a Calao Systems QIL-A9260 Board.
          <http://www.calao-systems.com>

config MACH_SBC35_A9260
        bool "CALAO SBC35-A9260"
        help
          Select this if you are using a Calao Systems SBC35-A9260.
          <http://www.calao-systems.com>

config MACH_FLEXIBITY
        bool "Flexibity Connect board"
        help
          Select this if you are using Flexibity Connect board
          <http://www.flexibity.com>

config MACH_LUX_SF9
        bool "Evo-Teh LUX-SF9 board"
        select HAVE_AT91_DATAFLASH_CARD
        select HAVE_NAND_ATMEL_BUSWIDTH_16
        help
          Select this if you are using Evo-Teh LUX-SF9 board based on the Atmel AT91SAM9260.

endif
```

Slika 32: Dodajanje nastavitev v datoteko Kconfig.

Sedaj nam samo še preostane definirati, katere izvorne datoteke se bodo prevedle in povezale v jedro. Temu opravilu pa je namenjena datoteka `Makefile` .

6.4.7.3 Datoteka Makefile

Ko prevajamo jedro, program make preverja konfiguracijo in se na podlagi tega odloča v katere mape mora še pogledati in katere izvorne datoteke prevesti. V našem primeru moramo določiti, katere datoteke se bodo prevedle, ko bomo v meniju izbrali ploščico LUX-SF9.

Specifično vhodno izhodno periferijo, ki je značilna za našo ploščico, smo opisali in definirali delovanje v izvorni datoteki `board-lux-sf9.c`, ki smo jo skopirali v podmapo `arch/arm/mach-at91`. Sedaj moramo dodati še ustrezne nastavitve v datoteko `Makefile` v isti podmapi (slika 33).

```
…
# AT91SAM9260 board-specific support
obj-$(CONFIG_MACH_AT91SAM9260EK) += board-sam9260ek.o
obj-$(CONFIG_MACH_CAM60)         += board-cam60.o
obj-$(CONFIG_MACH_SAM9_L9260)    += board-sam9-l9260.o
obj-$(CONFIG_MACH_USB_A9260)     += board-usb-a9260.o
obj-$(CONFIG_MACH_QIL_A9260)     += board-qil-a9260.o
obj-$(CONFIG_MACH_SBC35_A9260)   += board-sbc35-a9260.o
obj-$(CONFIG_MACH_AFEB9260)      += board-afeb-9260v1.o
obj-$(CONFIG_MACH_CPU9260)       += board-cpu9krea.o
obj-$(CONFIG_MACH_FLEXIBITY)     += board-flexibity.o
obj-$(CONFIG_MACH_LUX_SF9)       += board-lux-sf9.o
…
```

Slika 33: Definiranje objekta za vključitev v povezovalnik.

Kot lahko ugotovimo, je struktura te datoteke precej preprosta, kar je posledica popolne prenove sistema gradnje jedra iz verzije 2.4.x na 2.6.x, kjer so razvijalci posvetili veliko časa temu, da so poenostavili dodajanje dodatnih modulov in podpore. Na podoben način, smo podprli še ostale plošče in pa dodali gonilnike za dodatno strojno opremo.

6.5 Korenski datotečni sistem in distribucija Ångström

Vsak projekt, ki ima za cilj zagotoviti programsko podporo stojni opremi na osnovi vgrajenega operacijskega sistema Linux, mora pripraviti štiri osnovne komponente:

- Navzkrižno razvojno orodje (angl. toolchain) – vsebuje prevajalnik, povezovalnik in druga orodja za kreiranje in prevajanje programske kode. Vse je odvisno od orodij, opisanih v poglavju 6.2.4.
- Zaganjalnik (angl. bootloader) – potreben za osnovni zagon in inicializacijo ploščice ter nalaganje in zagon jedra Linux (poglavje 6.3).

- Jedro (angl. kernel) – srce sistema, ki upravlja s sistemskimi viri in služi kot vmesnik do strojne opreme (poglavje 6.4).
- Korenski datotečni sistem (angl. root filesystem) – vsebuje knjižnice in programe, ki se zaženejo, ko jedro zaključi svoj zagon.

V tem poglavju se bomo osredotočili na četrti in zadnji element vgrajenega sistema Linux korenski datotečni sistem, ki ga v našem primeru zagotavlja distribucija Ångström [24].

Distribucija Ångström je ena izmed distribucij Linux, namenjena vgrajenim sistemom. Zanjo je značilno, da poleg izgradnje slike korenskega sistema omogoča tudi uporabo upravljalnika programskih paketov opkg, preko katerega lahko na delujoč vgrajeni sistem namestimo dodatne programske pakete iz različnih skladišč programskih paketov (angl. repository). Opkg ali Open PacKaGe Management je enostaven upravljalnik programskih paketov, posebej namenjen uporabi v vgrajenih sistemih ravno zaradi svoje lahkotne in minimalistične zasnove. Pri razvoju so se zgledovali po upravljalniku dpkg, ki je značilen za distribucije Debian, tako da je sintaksa uporabe programa zelo podobna.

6.5.1 Kaj mora vsebovati korenski sistem?

Jedro Linux pridobi dostop do korenskega datotečnega sistema tako, da pripne (angl. mount) blokovno napravo (particija na disku ali pomnilniku flash), ki je definirana kot vhodni parameter `root=` pri zagonu. Ko dobi dostop do korenskega datotečnega sistema, jedro zažene prvi program z imenom `init`, kar smo podrobneje opisali v poglavju 6.4.6.2. S tem je za jedro delo končano. Sedaj mora program `init` poskrbeti, da se zaženejo drugi programi in skripte, ki potem kličejo sistemske funkcije definirane v sistemski knjižnici C in preko nje sistemske klice jedra.

Vsi Linux operacijski sistemi potrebujejo naslednji minimalni nabor komponent korenskega datotečnega sistema, da so uporabni:

- **init** – program, ki požene celoten sistem v delovanje, ponavadi tako, da zažene več skript;
- **ukazna lupina (angl. shell)** – omogoča interakcijo z ukazno vrstico in izvaja skripte, ki jih je pognal program `init`;
- **daemons** – strežniški programi, ki se izvajajo v ozadju in jih zažene `init`;
- **sistemske knjižnice (angl. libraries)** – programi uporabljajo skupne deljene knjižnice, ki morajo biti prisotne na sistemu;

- **konfiguracijske datoteke** – konfiguracijske datoteke programov so ponavadi shranjene v mapi `/etc`;
- **vozlišča naprav (angl. device nodes)** – posebne unix datoteke, ki omogočajo dostop do gonilnikov naprav na nivoju datotečnega sistema;
- **/proc in /sys** – dva psevdo datotečna sistema, ki predstavljata podatkovne strukture jedra kot hierarhično strukturo map in datotek;
- **moduli jedra (angl. kernel modules)** – določeni deli jedra, predvsem gonilniki se lahko prevedejo kot moduli in se ponavadi nahajajo v `/lib/modules/kernel/`.

Teoretično bi lahko vse te strukture združili v en program, vendar se to uporablja le v zelo redkih primerih, saj želimo omogočiti modularnost in razširljivost sistema. Zagotoviti pravilno delovanje in pa učinkovit, predvsem pa ponovljiv sitem izgradnje teh komponent je zelo zapleten proces, zato bomo pri pripravi korenskega datotečnega sistema uporabili avtomatiziran sistem za izgradnjo [13].

6.5.2 Sistem za izgradnjo OpenEmbedded

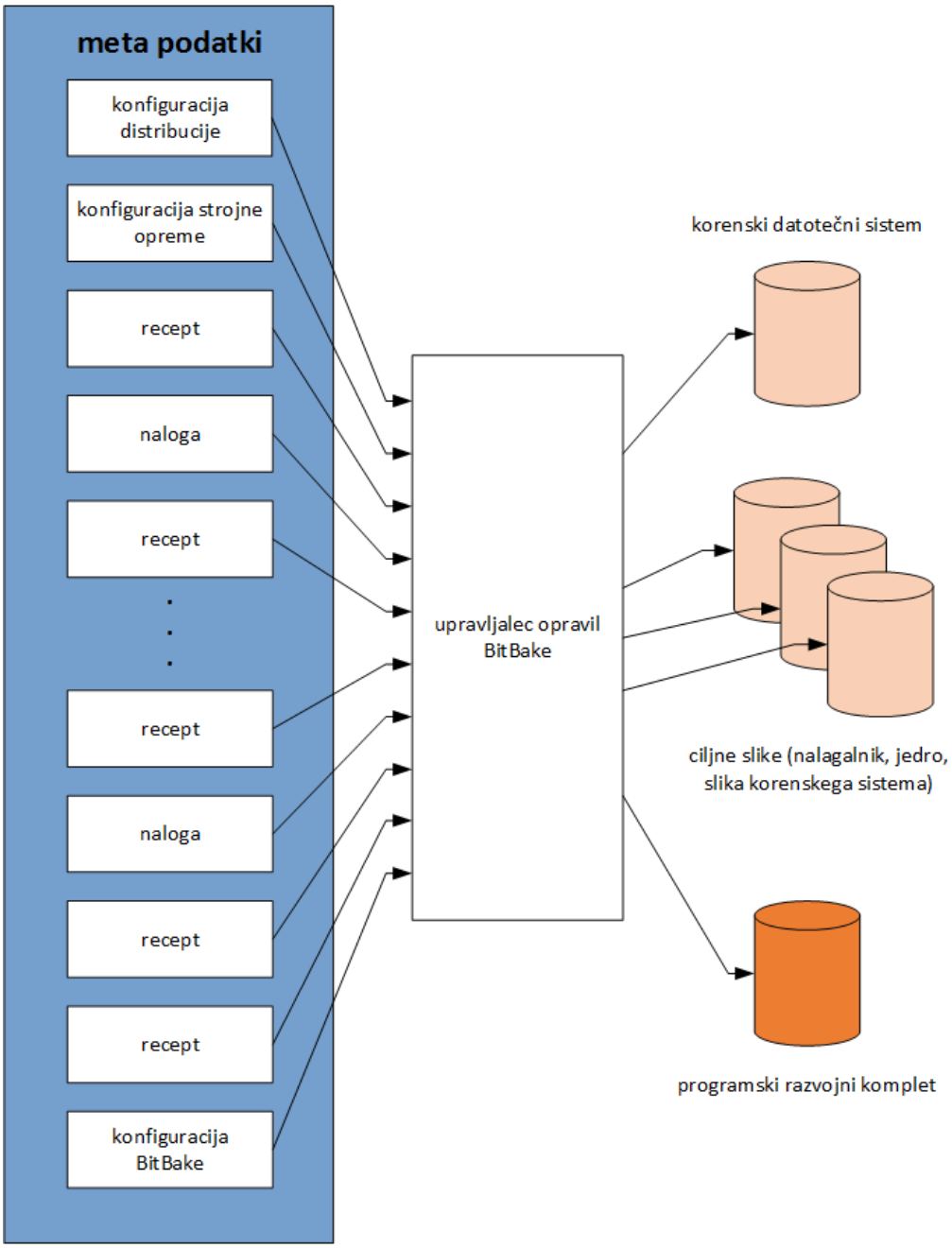
Sistem za izgradnjo (angl. build system) je skupek orodij in receptov, ki omogočajo avtomatizacijo izgradnje osnovnih komponent vgrajenega sistema Linux. Mi ga bomo uporabili za pripravo distribucije Linux, ki nam bo zagotovila osnovno korensko datotečno sliko. Prednost takšnega sistem je v ponovljivosti postopka, saj sistem sam rešuje medsebojne odvisnosti programskih paketov, zagotovi vključitev potrebnih popravkov, prevede programe in sestavi sliko korenskega datotečnega sistema, ki jo lahko prenesemo na strojno opremo.

Obstaja več odprtokodnih sistemov za gradnjo, eden najbolj znanih, ki ga bomo uporabili pri gradnji našega korenskega sistema in distribucije Linux je OpenEmbedded [25].

Projekt OpenEmbedded si je zadal cilj pripraviti binarne pakete z uporabo formata .ipk, ki jih lahko združimo in uporabimo na različne načine za izgradnjo tarčnega sistema. To so dosegli tako, da so napisali večje število za izgradnjo vsakega programa posebej. Pri izgradnji teh programov na podlagi receptov in pravil so uporabili upravljalca opravil BitBake. Ti recepti in navodila, če jih povežemo skupaj predstavljajo metapodatke, na podlagi katerih lahko sestavimo celotno distribucijo Linux. Skupek takšnih specifičnih metapodatkov je tudi distribucija Ångström.

OpenEmbedded je sestavljen iz dveh glavnih elementov, ki sta predstavljena na sliki 34. BitBake je močno in fleksibilno orodje za izgradnjo in je uporabljen kot upravljalec opravil.

Metapodatki pa predstavljajo skupek navodil, ki povedo programu BitBake kaj in kako sestaviti v neko celoto.



Slika 34: BitBake in metapodatki.

BitBake pregleda metapodatke, ki vsebujejo informacije kaj in kako prevesti programe. Rezultat procesa gradnje po navodilih definiranih v receptih je tako lahko korenski datotečni sistem, jedro, nalagalnik ali pa program. Ena izmed zanimivih možnosti uporabe receptov je, da se na takšen način pripravi celotno razvojno okolje (angl. software development kit, SDK),

ki vsebuje prevajalnik, knjižnice in izvorne datoteke, ki jih lahko potem uporabimo za razvoj aplikacij [11].

Metapodatke lahko razdelimo na štiri kategorije, vsaka z svojo točno določeno funkcijo:

- recepti (angl. recipes),
- razredi (angl. classes),
- naloge (angl. tasks),
- konfiguracije (angl. configuration).

Najbolj tipični tip metapodatkov so recepti. Recept je ponavadi samostojna datoteka ali pa majhna zbirka datotek, ki opisujejo kreiranje enega paketa. Napisani so v programskem jeziku Python ali pa skriptnem jeziku bash in imajo končnico `.bb`. Občutek za opisni jezik bomo dobili tako, da si pogledamo primer kreiranja preprostega paketa helloword_1.0.0.bb (slika 35).

```
DESCRIPTION = "Minimal statically compiled Hello world!"
LICENSE = "GPL"
PR = "r1"

S = "${WORKDIR}/${P}"

do_fetch () {
        mkdir -p ${WORKDIR}/${P}
        cd ${WORKDIR}/${P}
        printf "#include <stdio.h>\nint main(void)\n{\n\tprintf(\"Hello
world!\\\n\");\twhile(1);\n\treturn 0;\n}\n" >helloworld.c
}

do_compile () {
        ${CC} ${CFLAGS} -c -o helloworld.o helloworld.c
        ${CC} ${LDFLAGS} -o helloworld helloworld.o -static
}

do_install () {
        install -d ${D}${bindir}
        install -m 0755 helloworld ${D}${bindir}/
        # /bin/init is on purpose, it is tried after /sbin/init and /etc/init
        # so if a sysvinit is installed, it will be used instead of helloworld
        install -d ${D}${base_bindir}
        ln -sf ${bindir}/helloworld ${D}${base_bindir}/init
}
```

Slika 35: Preprost OpenEmbedded recept: helloworld_1.0.0.bb .

V primeru so uporabljene tri metode, ki prekrivajo privzete funkcije programa BitBake. Metoda `do_fetch()` kreira izvorno datoteko `helloworld.c`, metoda `do_compile()` jo prevede s privzetim prevajalnikom, metoda `do_install()` pa pokliče ukaz Linux `install`, ki kreira izhodno mapo ter vanjo kopira program. Na podoben način so pripravljeni recepti za vse ostale programe, ki jih definira določena distribucija Linux ali pa jih vključimo sami v korensko sliko sistema.

Vsi glavni ukazi fetch, unpack, configure, compile in install, ki se uporabljajo za kreiranje paketov so definirani v OpenEmbedded razredih in se nahajajo v datotekah s končnico `.bbclass`. Ena pomembnejših z vsemi privzetimi funkcijami je `base.bbclass`.

Naloge so posebna vrsta recepta, ki grupira posamezne med sabo odvisne pakete v neko celoto in se uporablja za definiranje korenskega datotečnega sistema določene distribucije Linux.

6.5.2.1 Konfiguracija sistema OpenEmbedded

Najtežji del uporabe sistema za gradnjo OpenEmbedded je priprava delujoče konfiguracije. Potrebno je pravilno definirati precej parametrov, da bi lahko zgradili vgrajeno distribucijo Linux. Najbolj pomembni parametri, ki jih moramo določiti so:

- ciljna arhitektura,
- tip procesorja,
- značilnosti ploščice ali modula, kot so serijska vrata, organizacija pomnilnika flash,
- izbira ustrezne knjižnice C (glibc, newlib, uclibc),
- prevajalnik in ostala binarna orodja,
- izhodni tip slike korenskega datotečnega sistema,
- verzija jedra Linux.

Seveda pa to niso edini konfiguracijski podatki za izgradnjo sistema. Konfiguracijske metapodatke lahko razdelimo na štiri večje skupine: BitBake, strojna oprema, distribucija in lokalni metapodatki. BitBake se upravlja preko datoteke `bitbake.conf`, vendar bomo zelo redko potrebovali kaj spremeniti, razen če ne zahtevamo od sistema kaj eksotičnega. Strojno opremo definiramo v svoji konfiguracijski datoteki, v našem primeru `./machine/lux-sf9.conf`. Osnovni parametri konfiguracije strojne opreme so prikazani na sliki 36.

```
#@TYPE: Machine
#@Name: LUX9 Development platform
#@DESCRIPTION: Machine configuration for the lux-sf9 development board with at91sam9260
processor
h a at91sam9260 processor

TARGET_ARCH = "arm"

PREFERRED_PROVIDER_virtual/kernel = "linux"
PREFERRED_VERSION_linux = "2.6.38"

KERNEL_IMAGETYPE = "uImage"

#don't try to access tty1
USE_VT = "0"

MACHINE_FEATURES = "kernel26 alsa ext2 usbhost usbgadget"

# used by sysvinit 2
SERIAL_CONSOLE = "115200 ttyS0"
IMAGE_FSTYPES ?= "ubi"
EXTRA_IMAGECMD_jffs2 = "--pad --little-endian --eraseblock=0x20000 -n"

# do ubiattach /dev/ubi ctrl -m 4
# From dmesg:
# UBI: smallest flash I/O unit:    2048
# UBI: logical eraseblock size:    129024 bytes
# from ubiattach stdout:
# UBI device number 0, total 1996 LEBs
MKUBIFS_ARGS = "-m 2048 -e 129024 -c 1996"

# do ubiattach /dev/ubi_ctrl -m 4
# from dmesg:
# UBI: smallest flash I/O unit:    2048
# UBI: physical eraseblock size:   131072 bytes (128 KiB)
# UBI: sub-page size:              512
UBINIZE_ARGS = "-m 2048 -p 128KiB -s 512"

require conf/machine/include/tune-arm926ejs.inc
```

Slika 36: Konfiguracijska datoteka strojne opreme lux-sf9.conf.

Konfiguracija distribucije definira vse vidike vgrajene distribucije Linux, ki jo želimo kreirati. Trenutna verzija okolja OpenEmbedded vsebuje nastavitvene datoteke za več kot 35 distribucij. Med njimi je najbolj znana ravno Ångström.

Zadnja kategorija konfiguracije pa je datoteka `local.conf`, kjer prilagodimo distribucijo našim potrebam. Minimalno moram definirati strojno opremo in pa izbrano distribucijo (slika 37), da ju sistem za izgradnjo poveže skupaj v celovit sistem.

```
DISTRO   = "angstrom-2008.1"
MACHINE ?= "lux-sf9"
```

Slika 37: Definirana distribucija in strojna oprema v lokalni konfiguracijski datoteki.

6.5.2.2 Kreiranje slike korenskega sistema

Svojo največjo moč pokažejo ravno recepti za kreiranje slik korenskega sistema. S pravilnim receptom je mogoče zgraditi celotno vgrajeno distribucijo Linux. Na voljo imamo več kot 100 receptov za različne slike korenskega datotečnega sistema, ki jih lahko prilagodimo svojim

potrebam. Ker ne uporabljamo sistema z grafičnim prikazovalnikom, smo se odločili uporabiti minimalno sliko korenskega sistema, ki omogoča prijavo v konzolo preko mreže ali serijskega vmesnika, vseeno pa nam omogoča, da namestimo dodatno pakete iz zunanjega podatkovnega skladišča. Ukaz, ki nam to omogoča je zelo preprost:

```
$bitbake base-image
```

Ta preprost ukaz zgradi korenski datotečni sistem, ki vsebuje minimalno zbirko paketov, potrebnih za zagon vgrajene naprave in omogoči dostop do ukazne vrstice. Navodila lahko seveda prilagodimo tako, da dodamo še druge programske pakete in s tem kreiramo distribucijo, ki ustreza našim potrebam.

Zadnji korak, ki nam sedaj preostane je, da vse komponente našega sistem, zaganjalnika AT91Bootstrap in U-Boot, jedro Linux ter sliko korenskega datotečnega sistema naložimo na ploščico in vključimo napajanje. Z malo sreče bi pred nami moral biti delujoč sistem, s katerim lahko komuniciramo preko serijskega vmesnika ali mrežne povezave. Od tukaj naprej pa lahko začnemo razvijati programsko opremo za reševanje problema, za katerega smo izbrali vgrajen sistem, saj imamo tudi že pripravljeno orodje za navzkrižni razvoj.

Poglavje 7 Sklepne ugotovitve

V diplomskem delu smo predstavili proces priprave paketa programske opreme in prenosa operacijskega sistema Linux za vgrajen sistem iz družine LUX9. Osredotočili smo se na štiri najbolj kritične elemente vgrajenega sistema Linux in sicer: orodja za razvoj programske opreme, zaganjalnik, jedro in korenski datotečni sistem. Omejili smo se na opis delov, ki so nujno potrebni za proces priprave paketa programske opreme.

Pred nami je bil izziv, kako na po meri razvito strojno opremo prenesti operacijski sistem Linux, skupaj s podporo različnim naborom perifernih naprav ter zagotoviti končnim uporabnikom produkta uporabo celotne funkcionalnosti naprave skupaj z razvojnim okoljem za aplikacije. Prva naloga vsakega razvijalca sistemske programske opreme je zelo dobro poznavanje stroje opreme, za katero je potrebno zagotoviti podporo. V našem primeru je bil to mikroprocesor na osnovi arhitekture ARM. Rezultat natančnega pregleda dokumentacije procesorja in razumevanje delovanja, predvsem vseh njegovih funkcionalnosti je razviden v poglavju 4. To znanje nam je bilo v veliko pomoč skozi celoten projekt, še posebej pa pri pisanju nizkonivojskih funkcij in programov za testiranje pravilnosti delovanja pomnilnika ter obeh zaganjalnikov. Najprej prvo stopenjskega, kjer je bilo potrebno inicializirati krmilnike pomnilnika skupaj z zunanjimi pomnilniki ter zagnati drugo stopenjski zaganjalnik. Tudi prilagoditev drugo stopenjskega zaganjalnika je terjala ker precej sprememb v programski kodi, za kar je nujno potrebno dobro poznavanje arhitekture, še posebej pri odločitvah, kje na različnih bliskovnih pomnilnikih se bodo nahajale posamezne komponente sistema.

Čeprav jedro Linux že v osnovi podpira arhitekturo ARM je potrebno pri izbiri mikroprocesorjev za projekte biti zelo oprezen. Podpora arhitekturi pomeni, da je v osnovi zelo dobro podprto samo jedro ali procesor. Ker imamo pri vgrajenih sistemih opravka s sistemi na čipu, pa potrebujemo poskrbeti tudi za podporo vseh ostalih komponent na čipu. Osnovnemu jedru Linux smo morali dodati, kar precej popravkov za mikroprocesorje AT91SAM9, da smo lahko zagotovili podporo vgrajenim komponentam. Na srečo obstaja skupina samostojnih razvijalcev pod okriljem podjetja Atmel, ki objavljajo te popravke. V prihodnosti se nadejamo, da bodo postali ti popravki del osnovne veje razvoja jedra Linux. Za analogno digitalni pretvornik, ki se uporablja za merjenje napetosti baterijskega napajanja, in vgrajeni krmilnik RS485 pa smo morali podporo in gonilnik zagotoviti sami. Naslednja nestandardna odločitev

je bila tudi uporaba gonilnika UBI in datotečnega sistema UBIFS za korenski datotečni sistem, ki skupaj omogočata boljše upravljanje z bliskovnim pomnilnikom. Za nas je bilo najbolj pomembno transparentno upravljanje z okvarjenimi bloki in sorazmerno razporejanje števila zapisov v posamezen blok. S tem smo zagotovili daljšo življenjsko dobo bliskovnega pomnilnika in možnost uporabe programske opreme, ki zahteva veliko vhodno izhodnih operacij.

Nekaj težav nam je povzročala tudi priprava slike korenskega sistema predvsem zaradi izbire datotečnega sistema UBIFS. Sistem za izgradnjo OpenEmbedded je zelo kompleksno orodje, ki omogoča avtomatizacijo celotnega procesa, seveda pa moramo zagotoviti pravilna navodila kako to narediti. Držali smo se načela, da pripravimo minimalno sliko datotečnega sistema tudi zaradi hitrejšega programiranja modulov. Ker smo izbrali vgrajeno distribucijo Ångström smo lahko dodatne programe namestili kasneje s pomočjo upravljalnika paketov opkg.

Projekt priprave programske podpore za družino LUX9 je trajal malo več kot eno leto in v tem času smo dobro spoznali operacijski sistem Linux in pridobili potrebno znanje za podporo novim modulom in ploščicam v prihodnosti. Več energije bi lahko vložili v avtomatizacijo določenih procesov. Za takšne namene je na voljo skupek orodij, združenih pod okrilje projekta Yocto. Ta omogoča izgradnjo vseh štirih komponent vgrajenega operacijskega sistema Linux znotraj enega okolja na še bolj strukturiran način. Zaradi svoje kompleksnosti pa uporaba tega programskega paketa zahteva še dodatne vložke v času in znanju.

Uspešnost projekta lahko ocenimo glede na izkušnje iz uporabe v realnem okolju. Po vsem svetu se trenutno nahaja 1000 enot vgrajenih sistemov temelječih na platformi LUX9, ki so v uporabi 24/7. Uporabniška spletna aplikacija, ki teče na operacijski programski opremi, poleg spletnega strežnika uporablja tudi podatkovno bazo SQLite. O kvaliteti in stabilnosti prenosa operacijskega sistema Linux priča dejstvo, da do sedaj ni bilo potrebno pripraviti popravka za jedro Linux ali za kakšnega od gonilnikov sistema.

Izvorna koda, ki je del tega projekta, je prosto dostopna na naslovu
https://github.com/davidpristovnik/LUX9 .

Literatura

[1] Bovet, Daniel P. and Cesati, Marco. Understanding the Linux Kernel, Third Edition. O'Reilly, 2006.

[2] Steve Furber. ARM System-on-Chip Architecture. Addison-Wesly, 2nd edition, 2000.

[3] John Goodacre and Andrew N. Sloss. Parallelism and the ARM instruction set architecture. IEEE Computer, 38(7):42-50, 2005.

[4] Hallinan, Christopher. Embedded Linux primer: a practical real-world approach, Second Edition. Prentice Hall, 2010.

[5] Kerrisk, Michael. The Linux programming interface: a Linux and UNIX System Programming Handbook. No Starch Press, 2010.

[6] Krazit, Tom, April 3, 2006. *ARMed for the Living Room*. Available: http://news.cnet.com/ARMed-for-the-living-room/2100-1006_3-6056729.html

[7] James A. Langbridge. Professional Embedded ARM Development. John Wiley & Sons, Inc., 2014.

[8] Love, Robert. Linux Kernel Development, Third Edition. Pearson Education, Inc, 2010.

[9] Ryzhyk, Leonid, June 5, 2006. The ARM Architecture.

[10] Gene Sally. Pro Linux Embedded Systems. Apress, 2010.

[11] Otavio Salvador and Daiane Angolini. Embedded Linux Development with Yocto Project. Packt Publishing, 2014.

[12] Andrew N. Sloss, Dominic Symes, Chris Wright with contribution by John Rayfield. ARM System Developer's Guide, Designing and Optimizing System Software. Morgan Kaufmann Publishers, 2004.

[13] Chris, Simmonds. Mastering Embedded Linux Programming. Packt Publishing, 2015.

[14] David Seal. ARM Architecture Reference Manual. Addison-Wesly, 2nd edition, 2000.

[15] Karim Yaghmour, Jon Masters, Gilad Ben-Yossef, and Philippe Gerum. Building Embedded Linux Systems, Second Edition, 2008.

[16] LUX Boards [Online; dostop 10.9.2015]. Dosegljivo: http://www.luxboards.com/foswiki/LuxBoards/Lux9Boards

[17] Atmel-6384F-ATARM-SAM9G20-Datasheet [Online; dostop 9.12.2015]. Dosegljivo: http://www.atmel.com/Images/Atmel-6384-32-bit-ARM926-Embedded-Microprocessor-SAM9G20_Datasheet.pdf

[18] Atmel-6221M-ATARM-SAM9260-Datasheet [Online; dostop 9.12.2015]. Dosegljivo: http://www.atmel.com/Images/Atmel-6221-32-bit-ARM926EJ-S-Embedded-Microprocessor-SAM9260_Datasheet.pdf

[19] Software Package with AT91LIB version 1.5 [Online; dostop 8.11.2013]. Dosegljivo: http://www.atmel.com/images/at91lib_20100901_softpack_1_5_svn_v8476.zip

[20] AT91SAM9260-EK Software Package for IAR 5.2, Keil and GNU [Online; dostop 8.11.2013]. Dosegljivo: http://www.atmel.com/images/at91sam9260-ek.zip

[21] Das U-Boot -- the Universal Boot Loader [Online; dostop 8.11.2013]. Dosegljivo: http://www.denx.de/wiki/U-Boot

[22] AT91 Linux 2.6 Patches [Online; dostop 8.11.2013]. Dosegljivo: http://maxim.org.za/at91_26.html

[23] AT91SAM Community [Online; dostop 8.11.2013]. Dosegljivo: http://www.at91.com

[24] The Ångström Distribution [Online; dostop 8.11.2013]. Dosegljivo: http://www.angstrom-distribution.org

[25] OpenEmbedded, the build framework for embedded Linux [Online; dostop 8.11.2013]. Dosegljivo: http://www.openembedded.org/wiki/Main_Page

[26] The Kernel Archives [Online; dostop 9.12.2015]. Dosegljivo: https://www.kernel.org/